Generació procedural de terrenys potencialment infinits utilitzant CUDA

Treball final de grau

Genís Solé

dirctor
Antonio Chica Calaf

Departament de Ciències de la Computació (CS)

Grau en Enginyeria en Informàtica, Especialitat Computació.

FACULTAT D'INFORMÀTICA DE BARCELONA (FIB)

UNIVERSITAT POLITÈCNICA DE CATALUNYA (UPC) – BarcelonaTech

23 d'octubre de 2017

Resum

En el món dels gràfics per computador és un problema recurrent la necessitat de generar i visualitzar terrenys. A vegades, aquests requereixen ser extremadament grans o inclús no finits. En aquest casos la única solució és la seva generació procedural sota demanda. En aquest projecte es presenta la implementació de una llibreria de caràcter general que proporciona els mecanismes bàsics tant per a la generació com la visualització d'aquests terrenys. Aquesta llibreria aprofita l'alt nivell de paral·lelisme dels algorismes de generació de terrenys basats en soroll fractal utilitzant el marc de còmput paral·lel en GPU CUDA. D'aqueta forma, no només s'acceleren aquest algorismes, sinó que les dades generades queden llestes per ser visualitzades, en aquest cas pel marc de renderizat OpenGL, evitant qualsevol tipus de transferència.

Resumen

En el mundo de los gráficos por computador un problema recurrente es la necesidad de generar y visualizar terrenos. A veces, estos requieren ser extremadamente grandes o incluso no ser finitos. En estos casos la única solución es su generación procedural bajo demanda. Nosotros presentamos la implementación de una librería de carácter general que proporciona los mecanismos básicos tanto para la generación como la visualización de estos terrenos. Esta librería aprovecha el alto nivel de paralelismo del los algoritmos de generación de terrenos basados en ruido fractal utilizando el marco de computo paralelo en GPU CUDA. De esta manera, no solo se aceleran estos algoritmos, sino que los datos generados quedan listos para ser visualizados, en este caso por el marco de renderizado OpenGL, evitando cualquier tipo de transferencia.

Abstract

The need of generation and visualization of terrains is a common problem in computer graphics. Sometimes these terrains require to be extremely large or even not finite. The only solution for these cases is on-the-fly procedural generation. We present the implementation of a general purpose library that provides basic mechanisms for both, generation and visualization of these terrains. The library takes profit of the high degree of parallelism fractal noise terrain generation algorithms have using the CUDA parallel GPU computation framework. Thus, not only the algorithms are accelerated, but the generated data is ready to be visualized, in this instance with the OpenGL render framework, avoiding any transfer.

Índex

1 Introducció **4**
- 1.1 Formulació del problema . . . 4
- 1.2 Abast . . . 4
  - 1.2.1 Generació de la geometria . . . 5
  - 1.2.2 Mecanismes de *shading* . . . 5
  - 1.2.3 Sistema de texturació . . . 5
- 1.3 Organització del document . . . 6

2 Generació del terreny **7**
- 2.0.1 Representació del terreny . . . 7
- 2.0.2 Funció de soroll de Perlin . . . 8
- 2.0.3 Soroll fractal . . . 14
- 2.0.4 Obtenció del vector normal de la superfície . . . 17

3 Gestió computacional del terreny **18**
- 3.1 Geometrical Clipmaps . . . 18
  - 3.1.1 Descripció General . . . 19
  - 3.1.2 Memòria cau . . . 22
  - 3.1.3 Visualització del terreny . . . 27

4 Texturació del terreny **36**
- 4.0.1 Aplicació de les Wang Tiles . . . 36
- 4.0.2 Generació de un conjunt de rajoles . . . 42

5 Diseny i ús de la libreria **45**
- 5.0.1 TerrainRenderer . . . 46
- 5.0.2 TerrainModel . . . 49
- 5.0.3 TerrainDataCache . . . 51
- 5.0.4 Aplicació de Mostra . . . 51
- 5.0.5 Resultats . . . 54

6 Conclusió **57**

7 Informe de sostenibilitat **59**
- 7.1 Estudi de l'impacte econòmic . . . 59
- 7.2 Estudi de l'impacte social . . . 60
- 7.3 Estudi de l'impacte ambiental . . . 61

8 GEP **62**
8.1 Introducció . . . 62
8.2 Formulació del problema . . . 62
8.3 Abast . . . 62
8.3.1 Generació de la geometria . . . 63
8.3.2 Texturat . . . 63
8.3.3 CUDA . . . 64
8.4 Metodologia i rigor . . . 64

I Planificació del projecte **65**
8.1 Introducció . . . 66
8.2 Aprenentatge i recerca . . . 66
8.2.1 Recerca sobre la generació procedural de terrenys (1) . . . 66
8.2.2 Recerca sobre la generació de textures acícliques (2) . . . 66
8.2.3 Adquisició de coneixements de CUDA (3) . . . 66
8.3 Adaptació de les tècniques a les necessitat del projecte . . . 66
8.3.1 Adaptació de les tècniques de generació de terrenys escollides (4) . . . 67
8.3.2 Adaptació de les tècniques de generació de textures escollides (5) . . . 67
8.4 Primer prototipus . . . 67
8.4.1 Programació de un visualitzador simple (6) . . . 67
8.4.2 Programació de una primera versió de la generació de terrenys (7) . . . 67
8.4.3 Programació de una primera versió de la generació de textures (8) . . . 67
8.4.4 Finalització de la primera etapa (9) . . . 68
8.5 Avaluació dels resultats preliminars . . . 68
8.5.1 Avaluació del rendiment del primer prototipus (10) . . . 68
8.5.2 Avaluació del resultat visual del primer prototipus (11) . . . 68
8.5.3 Refinament del primer prototipus (12) . . . 68
8.6 Versió final . . . 69
8.6.1 Versió final del visualitzador (13) . . . 69
8.6.2 Versió final de la generació de terreny (14) . . . 69
8.6.3 Versió final de la generació de textures (15) . . . 69
8.7 Redactat de la memòria (16) . . . 69
8.8 Diagrama de dependències . . . 70
8.9 Diagrama de GANTT . . . 71

II Pressupost i sostenibilitat **72**
8.1 Introducció . . . 73
8.2 Identificació dels costs . . . 73
8.3 Estimació dels costos . . . 73
8.3.1 Costos de la oficina . . . 73
8.3.2 Equips . . . 74
8.3.3 Material . . . 74
8.3.4 Salaris . . . 74
8.3.5 Imprevistos . . . 74

8.3.6 Agregat del projecte 74
8.4 Viabilitat econòmica 75
8.5 Impacte social i ambiental 75
8.6 Rendició de costos i control 75

III Context i bibliografia **77**
8.1 Introducció 78
8.2 Context 78
8.2.1 Generació procedural de terrenys en els videojocs 78
8.3 Estat de l'art 78
8.3.1 Generació de la geometria 79
8.3.2 Texturació aciclica 80
8.3.3 CUDA 80

Capítol 1

Introducció

1.1 Formulació del problema

En el món dels gràfics per computador sovint es necessari visualitzar escenes exteriors. Un dels components principals d'aquestes escenes és el terreny el qual, en segons quines aplicacions, pot representar extensions d'espai molt grans. Per exemple, en el camp dels vídeojocs es pot voler representar un món on el jugador té una gran llibertat de moviments o inclús pot explorar indefinidament. Les dades que representen aquests elements orogràfics no és viable mantenir-les de forma persistent i, per tant, la única solució és la generació procedural d'aquestes sota demanada.

Per tal de poder construir aquests models de una forma prou realista una de les opcions més utilitzades són algorismes basats en soroll fractal. Algunes variacions d'aquest algorismes, tot hi que no estan consolidades directament en en els factors ambientals físics, simulen les formes que els processos naturals donen a la orografia real. Aquests algorismes es basen en la generació de mapes (o camps) d'alçades on cada mostra pot ser computada independentment de les altres i, per tant, aquesta tasca és extremadament paral·lelitzable cosa que motiva a la utilització de alguna plataforma paral·la, en aquest cas CUDA (Compute Unified Device Architecture).

Com hem dit abans es busquen terrenys amb un cert realisme, per tant, la posibilitat d'aplicar textures a aquest terrenys és un punt clau.

1.2 Abast

El projecte consisteix en la creació de una llibreria de caràcter general que pugui ser utilitzada per a crear aplicacions on es pugui navegar per terrenys generats proceduralment. Aquests terrenys haurien de ser potencialment infinits o quasi-infinits, és a dir, l'usuari pot explorar l'entorn en una direcció arbitraria de forma indefinida i, per tant, degut a que tenim recursos finits, la generació d'aquests s'ha de realitzar sota demanada. A més, aquesta llibreria ha de proporcionar mecanismes de il·luminació o *shading* prou genèrics per tal de no forçar a l'usuari a utilitzar cap model o estil concret. Entre aquest mecanismes de *shading* cal remarcar la importància de un sistema de texturació, ja que, serà amb aquest que l'usuari especificarà les dades de la superfície necessàries per a calcular la il·luminació (per exemple: color, rugositat, normals locals, etc).

A continuació, en les següents seccions, descriurem amb més detall els tres aspectes principals que engloben la totalitat del projecte i les tecnologies que emprarem per a la creació de les mateixes. Aquests aspectes són: la generació de la geometria, els mecanismes de *shading* i el sistema de texturació.

1.2.1 Generació de la geometria

Aquesta part del projecte consisteix en generar una representació de la geometria que forma el terreny de manera que es pugui visualitzar i explorar en temps real (típicament, superior a 30 imatges per segon) amb maquinari relativament modern. Aquesta representació, com hem esmentat anteriorment, ha de poder ser generada a mesura que la càmera és mou de forma que sembli que es pot avançar per l'escena indefinidament. Per a realitzar aquesta tasca s'utilitzaran diferents funcions de generació de soroll.

Per tal de garantir la fluïdesa en la interacció amb l'escena, la representació del terreny ha de ser capaç de filtrar el detall de les porcions d'aquest que es trobin allunyades de la càmera (o on el detall no sigui útil). D'aquesta manera es disminueixen notablement el nombre de mostres a computar i visualitzar i, per tant, es redueix el cost (tant temporal com d'emmagatzematge) d'aquestes operacions. A més, els algorismes més típics de generació procedural de terrenys tenen la característica de que permeten computar cada una de les mostres de forma independent i, per tant, de forma paral·lela. Això ens porta a voler explotar aquesta propietat utilitzant algun sistema de computació paral·lela basat en GPU (Graphic Prcessor Unit), concretament CUDA. D'aquesta manera no només accelerem la generació sinó que també ens estalviem la transferència de dades entre la CPU (Central Processing Unit) i la GPU la qual pot tenir un cost temporal no menyspreable. Cal afegir que CUDA és una tecnologia propietària de Nvidia i, per tant, només suportarem GPUs d'aquesta marca.

1.2.2 Mecanismes de *shading*

Una de les característiques clau de la llibreria és la de no generar imatges del terreny amb el color final sinó imatges amb una representació intermèdia amb les dades corresponents a cada pixel necessàries per computar, en una segona passada, la il·luminació d'aquest. Aquesta tècnica, coneguda amb el nom de *deferred shading*, normalment és utilitzada per a separar els càlculs geomètrics de l'escena dels càlculs de la il·luminació, d'aquesta manera la llibreria s'encarregarà de la part geomètrica mentre que l'usuari haurà de proporcionar la part de la il·luminació. Per generar aquesta imatge amb les dades geomètriques utilitzarem la API (Aplication User Interface) OpenGL i, per tant, el resultat serà presentat en forma de textura gestionada per aquest marc en algun dels formats que proporcioni. És per això que és molt recomanable que la part de la il·luminació també sigui desenvolupada amb la mateixa API.

1.2.3 Sistema de texturació

Com hem dit abans, un sistema de texturació és imprescindible per a garantir prou flexibilitat a l'usuari a l'hora d'especificar les dades necessàries per a calcular el color final de la superfície del terreny. Una de les tècniques de texturació més utilitzades per cobrir extincions de superfície no limitades és la de enrajolat o *tiling*. Aquesta tècnica consisteix en repetir una mateixa textura per tal de cobrir la superfície. Aquesta textura, tot hi que és creada de forma que el contingut

sigui continu entre dos instàncies posades una al costat de l'altra, genera patrons visuals molt notoris que podrien disminuir la qualitat visual de la imatge final. És per això que utilitzarem una tècnica basada en el *tiling* anomenada *wang tiling* que intenta incorporar més variació a l'enrajolat per tal d'evitar aquest patrons no desitjats. Aquesta tècnica requereix la creació de un conjunt de textures que tinguin unes propietats concretes, és per això que proporcionarem una aplicació apart que permet sintetitzar conjunts de textures vàlids a partir de una imatge font. Cal mencionar que a l'usuari no se li obliga a utilitzar aquesta aplicació si és capaç de generar amb els seus medis un conjunt de textures amb les propietats imposades per la tècnica.

1.3 Organització del document

El document està organitzat de la següent forma:

- En el capítol 2 mostrem com implementar diferents algorismes basats en soroll fractal per a generar terrenys i la seva representació computacional.
- En el capítol 3 presentem la tècnica de visualització de terrenys que hem escollit per a la realització d'aquest projecte. En podem distingir 2 parts: per una banda l'actualització de les dades que formen el terreny i per l'altra la visualització d'aquestes.
- En el capítol 4 presentarem un esquema per a poder texturar el terreny generat basat en *wang tiles*. Hi trobarem com sintetitzar un atles de rajoles de wang a partir de una imatge font i com aplicar-los a la geometria.
- En el capítol 5 es mostren alguns dels detalls del disseny i l'ús de la llibreria que hem creat. També mostrarem els resultats de una petita aplicació de prova que hem desenvolupat.
- En el capítol 6 concloem el document fent una mica de repàs de les tècniques utilitzades i apuntant el possible treball futur.
- En el capítol 7 presentem l'estudi de sostenibilitat del projecte.

Capítol 2

Generació del terreny

La generació computacional de terrenys és un camp important en el món dels gràfics per computador. La geometria d'aquest terrenys és un element clau en aplicacions que requereixin d'escenes a l'aire lliure com poden ser simulacions o videojocs. Aquests són objectes geomètricament molt complexos i per tant crear-los a mà pot arribar a ser inviable. És per això que hi ha certa motivació en trobar formes de generar-los de forma automàtica.

Existeixen multitud de tècniques per a construir geometria ressemblant a estructures orogràfiques. Nosaltres, tot hi que ens agradaria tenir cert realisme, ens allunyarem de les tècniques que simulen els processos naturals al detall i ens centrarem en una família de funcions basades en soroll coherent que, tot hi que no estan basades en els mecanismes naturals que esculpeixen els terrenys, proporcionen algunes de les característiques que podem trobar en aquests. A més, els terrenys que volem representar no presentaran estructures 3D com poden ser coves o saltants la qual cosa en facilitarà la representació computacional.

En aquest capítol mostrarem la representació computacional dels terrenys que utilitzarem juntament amb totes aquelles funcions necessàries per a la implementació dels mètodes que emprarem per a construir la representació lògica d'aquest terrenys.

2.0.1 Representació del terreny

Abans de introduir els procediments generadors de terreny és important conèixer la representació lògica que utilitzarem d'aquests. Aquesta representació ha de complir certs requisits que enumerem a continuació:

- Ha de poder ser generada en la GPU amb un marc de còmput paral·lel , especialment CUDA, de forma prou directa i eficient.
- Cada mostra ha de poder ser calculada independentment de les altres.
- Existeixin formes eficients de filtrar el detall on aquest no sigui útil.
- Ha de ser eficient visualitzar-la amb una API de renderització com OpenGL.

Una de les representacions digitals més típiques dels terrenys que compleix els requisits que hem establert anteriorment, i la que utilitzarem per la seva simplicitat i bon comportament amb

tècniques de generació sota demanda com la nostra, és la de els camps d'altures. Aquesta representació consisteix en en una graella 2D d'altures mostrejades a intervals regulars sovint guardada en forma de imatge amb un únic canal on cada pixel correspon a una d'aquestes mostres. Una observació que podem fer, és el fet de que les coordenades planars venen implícites en les coordenades de la imatge i, per tant, suposa un avantatge evident a nivell de emmagatzemament respecte un núvol de punts 3D que representa la mateixa superfície. A més, l'estructura regular facilita la reducció del nivell de detall, cosa que, com veurem més endavant, serà imprescindible per a poder visualitzar-la de forma eficient. Una altra de les avantatges d'aquesta regularitat és el fet de que facilita operacions d'intersecció amb el terreny que representen. Com a contrapartida és fàcil veure que aquesta forma de descriure superfícies no permet estructures 3D. En el nostre cas això no és un problema ja que els terrenys que volem representar no presenten aquest tipus d'estructures com podrien ser coves o saltants.

Existeixen altres possibles representacions, per exemple les triangulacions irregulars que intenten aproximar el terreny amb el detall representatiu de cada zona. Aquestes altres representacions les desestimem ja que, com hem dit abans, els camps d'altures s'adapten perfectament a les nostres necessitats i per tant la seva elecció és clara. A més, aquest tipus de representació és una de les més utilitzades, així que cal esperar trobar bones referències a l'hora de visualitzar-la de forma eficient.

2.0.2 Funció de soroll de Perlin

Una de les famílies de funcions generadores de terrenys més utilitzades, i la que hem escollit per el projecte, utilitza funcions de soroll coherent basades en graelles com a nucli de les seves operacions. Aquesta coherència ve donada per el fet de que punts propers en l'espai tenen valors similars, mentre que valors de punts distants es comporten de forma totalment independent. Per tal de que aquestes funcions siguin útils han de complir certs requisits que enumerarem a continuació:

- La funció de soroll ha de ser pseudoaleatòria, és a dir, donat una entrada, sempre obtindrem la mateixa sortida.
- Ha de tenir un rang conegut, en aquest cas, de -1 a 1.
- Ha de tenir una freqüència de 1.
- No ha de presentar patrons regulars obvis.
- Ha de ser estacionaria, és a dir, el seu caràcter estadístic no varia sota translació.
- Ha de ser isotròpica, és a dir, el seu caràcter estadístic no varia sota rotació.

Nosaltres utilitzarem la funció dissenyada per Ken Perlin coneguda com a *Perlin noise* i que va ser presentada per primer cop a [Perlin85]. Els motius d'aquesta elecció rauen principalment en la qualitat del soroll que genera la funció i a la senzillesa d'aquesta. Cal dir que exigeixen altres funcions de soroll com per exemple *value noise* o *simplex noise* que també solen ser utilitzades en la generació procedural de terrenys. Aquestes dos alternatives són més eficients que l'escollida. En el cas de *value noise* la qualitat del soroll generat no és gaire bo, produeix artefactes molt quadriculats. El *simplex* podríem dir que és superior a l'opció escollida en tot els aspectes. L'única raó per el que no l'hem utilitzat és per ser lleugerament més complicada i així preferint

la simplicitat de la funció utilitzada.

La definició de l'algorisme (*Perlin noise*) que defineix la versió 2D de la funció en un punt $p = (x, y)$ és el següent:

1. Per cada vèrtex en la graella 2D entera amb espaiat regular i unitari, es genera un vector 2D amb components en el rang [-1 i 1] pseudoaleatori r.

2. Per cada un dels vèrtex v de la cel·la on resideix p es computa el producte escalar entre el vector dels vèrtexs que em computat a (1.) i el vector $p - v$, és a dir, $d = r \bullet (p - v)$

3. S'interpolen amb una corba suau els 4 valors obtinguts a (2.) segons la posició de p en la cel·la.

Creiem oportú aclarir alguns dels punts anteriors. En primer lloc, en el punt (1.) no cal generar tots els vectors pseudoaleatoris de cop, només és necessari computar el dels vèrtexs que formen la cel·la on es troba p i que haurem de obtenir igualment en el pas (2.). Per trobar els vèrtex d'aquesta cel·la utilitzarem la següent expressió:

$$\begin{aligned} v_{00} &= floor(p) \\ v_{10} &= v_{00} + (1, 0) \\ v_{01} &= v_{00} + (0, 1) \\ v_{11} &= v_{00} + (1, 1) \end{aligned} \tag{2.1}$$

On v_{xy} denota les coordenades del vèrtex respecte respecte $floor(p)$ i $floor()$ denota la part entera (amb arrodoniment cap a $-\infty$).

Un altre punt que cal aclarir és el de la interpolació de (3). Aquesta corba típicament ve descrita per el polinomi de nivell 3 següent:

$$u(x) = 3x^2 - 2x^3 \tag{2.2}$$

O el de nivell 5:

$$u(x) = 6x^5 - 15x^4 + 10x^3 \tag{2.3}$$

Aquestes funcions tenen la propietat de que la seva derivada en els punts $x = 0$ i $x = 1$ és nul·la. A més, la de nivell 5 té l'avantatge de que la seva segona derivada també és nul·la. Això és important ja que, altrament, les derivades de segon grau presentarien discontinuïtats que podrien manifestar-se en el càlcul de la il·luminació.

el valor h resultat de la interpolació el podem calcular d'aquesta forma:

$$\begin{aligned} h = mix(&mix(d_{00}, d_{10}, u(fract(p_x))), \\ &mix(d_{01}, d_{11}, u(fract(p_x)), u(fract(p_y)) \end{aligned} \tag{2.4}$$

On $mix(x, y, a)$ ve definida per:

$$mix(x, y, a) = x(1 - a) + ya \tag{2.5}$$

, $fract(x)$ representa la part fraccionar de x i es pot definir de la següent forma:

$$fract(x) = x - floor(x) \tag{2.6}$$

i d_{xy} representa el resultat del producte escalar calculat a els vèrtexs v_{xy}.

Una petita optimització que podem considerar consisteix en expandir la expressió 2.4. D'aquesta forma, després de simplificar tot el possible, obtenim la següent expressió:

$$h = c_0 + u(fract(p_x)c_1 + u(fract(p_y))c_2 + u(fract(p_x))u(fract(p_y))c_3 \tag{2.7}$$

On:

$$\begin{aligned} c_0 &= d_{00} \\ c_1 &= d_{10} - d_{00} \\ c_2 &= d_{01} - d_{00} \\ c_3 &= d_{11} - d_{10} - d_{01} + d_{00} \end{aligned} \tag{2.8}$$

En la figura 2.1 podem veure un exemple de la clase de soroll que genera aquesta funció.

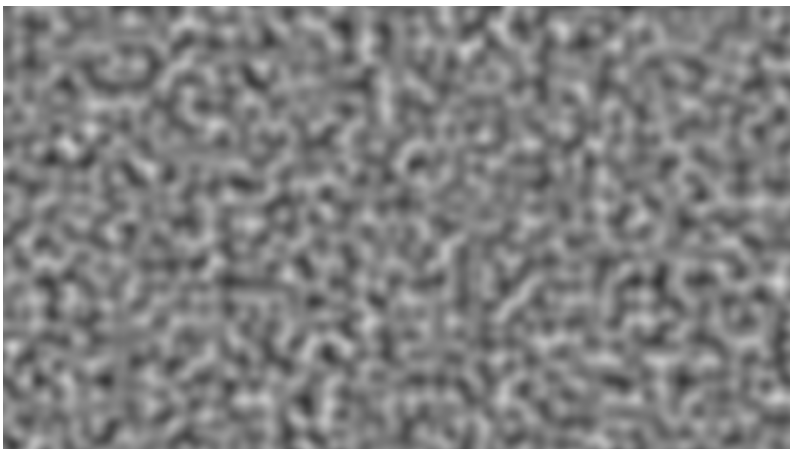


Figura 2.1: Mostra de Perlin noise on les regions fosques representen valors baixos i les regions clares valors alts.

Una propietat volguda d'aquest algorisme és que el valor de un punt pugui ser computat independentment dels altres. Això, com es pot veure, depen exclusivament de si els vectors pseudoaleatoris es poden computar independentment o no. Típicament els generadors de nombres pseudoaleatoris utilitzen seqüències de forma que el resultat següent depen de l'anterior o de l'estat en que aquedat el generador després de generar-lo. És clar que aquest tipus de generadors no compleixen amb els nostres requisits i per això queden descartats. Una altra solució són la de funcions de *hash*. En general, aquest tipus de funcions tenen com a objectiu manipular els elements d'entrada de forma que a la sortida no tingui cap relació amb el seu valor inicial al mateix temps que intenta minimitzar la correlació entre les entrades i les sortides. Com es pot veure, aquest és exactament el comportament que busquem per a generar el nostres vectors aleatoris. L'entrada seria les coordenades del vèrtex al que volem calcular el vector i la sortida hauria de ser un vector sense cap relació amb les coordenades entrades. Una forma de implementar aquest tipus de funcions es a través de taules de permutacions. En primer lloc es computen dos taules de forma aleatòria amb els mètodes seqüencials que hem descartat anteriorment, o amb mètodes que generin nombres aleatoris reals. Una d'aquestes taules conte un conjunt V de valors de sortida, l'altra un conjunt I de índexs. Sense entrar en detalls, la idea consisteix en indexar amb el valor d'entrada al conjunt I per obtenir un índex a un valor de sortida de V. Ho veurem més clar amb un exemple. Sigui

$q = (x, y)$ una coordenada de la que volem computar el seu vector. Un forma de calcular-ne l'índex i podria ser la següent:

$$i = I[(y + I[x \pmod{|I|}]) \pmod{|I|}] \pmod{|V|} \tag{2.9}$$

On l'operador $I[j]$ indica accés a l'element j del conjunt I. Cal notar que tant x com y han de ser enters i si no fos així, s'hauria de fer algun tipus de conversió prèvia. Ara, per obtenir el valor final només hem d'obtenir el valor de $V[i]$. També cal dir que els valors continguts a I poden ser més grans que $|I|$ i/o $|V|$, és per això que es realitza l'operació mod amb la cardinalitat del conjunt abans de indexar-lo. Tot hi que aquest tipus d'esquemes funcionen bé en entorns paral·lels com el que volem utilitzar, haver de llegir de memòria múltiples vegades de forma no coherent pot suposar un petit cost que podríem intentar reduir utilitzant funcions de *hash* totalment computacionals. La majoria d'aquestes funcions manipulen els bits que representen les entrades per tal d'aconseguir un comportament similar al descrit abans. D'aquesta forma no tindrem la flexibilitat de poder descriure la sortida en la forma necessària per resoldre el problema con fèiem amb V, sinó que el resultat serà una cadena de bits aleatòria la qual haurem de manipular per tal de que representi els valors que ens interessin. Una apreciació que podem fer és que la codificació de l'entrada és indiferent. En aquest cas les coordenades dels vèrtex de la graella entera poden tan ser codificades com enters de 32 bits en complement a 2, com per nombres decimals representats amb l'estàndard IEEE 754 de 32 bits (més conegut com a punt flotant) ja que seran tractades com a cadenes de bits que representen de forma única un element de un conjunt. Existeixen múltiples algorismes que poden crear cadenes pseudoaleatòries a partir d'altres cadenes i qualsevol d'aquest serviria. Nosaltres hem utilitzat una iteració de una de les funcions que es presenta a [Jenkins09] anomenada *One-at-a-Time Hash* però no hi ha cap motiu per no canviar-la en un futur si en trobem una de més eficient. Aquesta funció de hash la podem descriure amb el següent algorisme per a cadenes de bits de llargada 32:

Algorisme 1 Una iteració de la funció One-at-a-Time Hash per a cadenes de 32 bits

function HASH(x)
  $x \leftarrow x + (x << 10)$
  $x \leftarrow x \wedge (x >> 6)$
  $x \leftarrow x + (x << 3)$
  $x \leftarrow x \wedge (x >> 11)$
  $x \leftarrow x + (x << 15)$
  return x
end function

On l'operador $a << b$ indica desplaçar tots el bits de a, b posicions cap a la esquerra, l'operador $a >> b$ indica desplaçar totes el bits de a, b posicions cap a la dreta. En els dos casos els bits que queden en posicions superiors a 31 o inferiors a 0 s'eliminen i el bits que entren, ja que les cadenes resultants sempre son de 32 bits, tindran el valor de 0. L'operador $a \wedge c$ indica la operació `XOR` bit a bit entre la cadena de bits a i c i l'operador + indica la suma aritmètica d'enters sense signe.

És possible que si realitzéssim un anàlisis estadístic exhaustiu del comportament aleatori d'aquesta funció de hash descobriríem que realment no és gaire bona a nivell de la correlació que hi pugi haver entre l'entrada i la sortida. Afortunadament en el nostre cas només necessitem una certa percepció visual de aleatorietat. Com podem veure en la figura 2.2 la funció presenta una

variabilitat prou alta i no presenta patrons identificables a simple vista la qual cosa es suficient per a les nostres intencions.

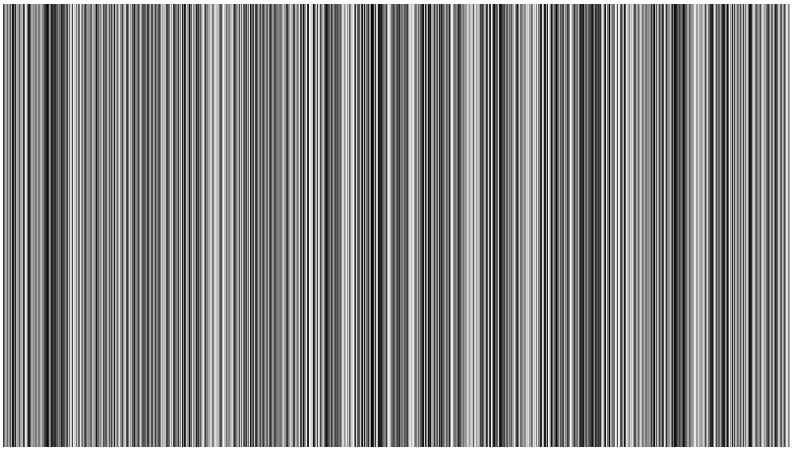


Figura 2.2: Mostra del resultat de aplicar la funció de hash presentada a l'algorisme 1 al rang de valors [-400, 400] en increments de una unitat per columna de pixels on el resultat es interpretat com a valors en el rang [-1, 1]. Les regions fosques representen valors baixos i les regions clares valors alts.

Com hem vist, a partir de dos coordenades (les de la graella) hem de obtenir un vector també format per a dos components. Per això haurem de crear dos cadenes de bits, una per a cada component del vector final, de forma que en la construcció d'aquestes dos cadenes hi intervinguin les dos coordenades anteriors. Les dos coordenades obtingudes, tot hi que parteixen dels mateixos valors, haurien de ser perceptualment independents entre elles. Una forma de fer-ho és la següent:

$$\begin{aligned} c_x &= h(y + h(x)) \\ c_y &= h(y + k + h(x)) \end{aligned} \tag{2.10}$$

On c_x i c_y són les cadenes aleatòries respectives del vector, x i y són les coordenades a la graella, $h()$ és la funció de hash presentada a l'algorisme 1 i k és una constant tal que $k \neq 0$. També cal dir que la suma aritmètica es podria substituir per a qualsevol operador que combinés dos cadenes de bits generant-ne una de nova de la mida estipulada de forma que representés d'alguna forma les dos cadenes inicials. En la figura 2.3 podem veure el resultat d'aplicar aquest esquema en una graella.

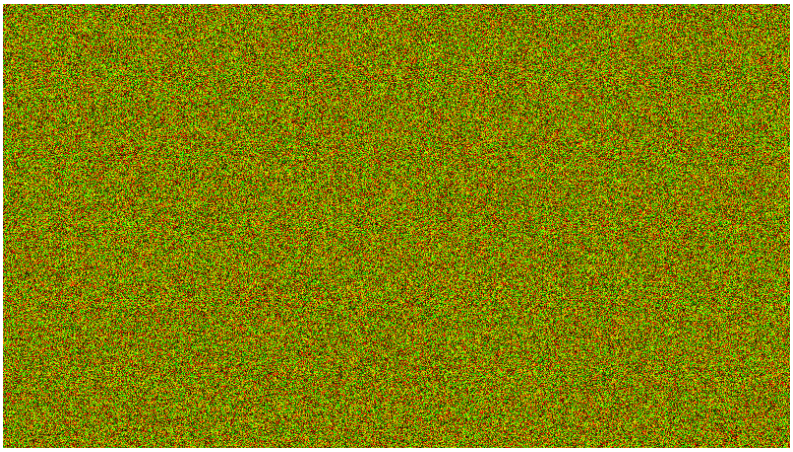


Figura 2.3: Mostra del resultat d'aplicar l'equació 2.10 a una graella de dimensions 800×450 en el rang [-400,400] i [-225, 225] respectivament, on cada un des vèrtex correspon a un pixel. El resultat està codificat en les components vermella (c_x) i la verda (c_y) del pixel.

Un cop generada la cadena de bits aleatoris, caldrà manipular-la per què representi un valor amb el que treballar. En el nostre cas volem obtenir un valor entre -1 i 1 representat amb l'estàndard IEEE 754 de 32 bits. Per fer això hem utilitzat el mètode descrit a [Quilez05]. En aquest article s'aprofita la disposició dels bits que descriu l'estàndard i que podem veure a continuació:

$$\underbrace{b_{31}}_{signe}\ \underbrace{b_{30}\dots b_{23}}_{exponent}\ \underbrace{b_{21}\dots b_0}_{mantissa} \tag{2.11}$$

On el valor representat per aquesta seqüència de bits ve donat per:

$$v = s \times 2^{e-127} \times m \tag{2.12}$$

On s és el signe (1 si el bit $b_{31} = 0$, -1 altrament), e és el valor codificat en complement a 2 a els bits d'exponent i m és el valor decimal binari representat per la cadena de bits $1.b_{21}\dots b_0$ (1 (punt) *mantissa*). Arribats a aquest punt, podem adonar-nos que m representa un nombre en el rang $[1, 2)$ aleatori. Recordem que partim de una cadena de bits aleatòria. Podem observar que si posem el bit de signe a 0 (positiu) i forcem $e = 128$ obtindrem valors en el rang $[2, 4)$ ja que en aquest cas $v = 2 \times m$. Ara si volem el valors en el rang $[-1, 1]$ només caldrà restar el valor per 3. Noteu que aquesta resta s'ha de realitzar amb aritmètica de punt flotant i, per tant, caldrà reinterpretar els bits en aquest format després de manipular-los. Una forma de fer-ho és la següent.

Algorisme 2 Conversió de una cadena de 32 bits aleatòria en un nombre representat amb l'estandard IEEE 754 de 32 bits en el rang [-1, 1].

function toFloat11(x)
  $x \leftarrow x >> 9$
  $x \leftarrow x | 0x40000000$
  return $r(x) - 3.0$
end function

On x representa una cadena de 32 bits representat un enter sense signe, l'operador $a >> b$ és el mateix que hem vist en la funció de hash, l'operador $a|b$ indica la `OR` bit a bit entre a i b,

els nombres que comencen amb $0x$ representen una cadena de bits expresada en hexadesimal i la funció $r()$ reinterpreta la cadena de bits que representa un enter sense signe a un nombre en punt flotant. Noteu que la cadena no canvia de cap manera, simplement la interpretació dels bits que la conformen. Podem veure que el desplaçament de bits posarà a 0 el 9 bits de més pes, els corresponents al signe i a l'exponent. A continuació amb la `OR` forçarem $e = 128$, és a dir, posarem a 1 el bit 30. Recordem que 128 es representa en complement a 2 com una cadena de 8 bits on només el de més pes té el valor 1.

2.0.3 Soroll fractal

Quan es busca en la literatura sobre la generació computacional de terrenys la primera referència que es troba és la utilització de funcions 2 dimensionals basades en soroll fractal o fBm (*fractional Brownian motion*), si utilitzem el terme estadístic. Existeixen diferents formes de generar i manipular aquests tipus de funcions per tal de crear formes ressemblats a la orografia que podem trobar als planetes. Aquestes funcions, en general, no tenen res a veure amb els processos físics que esculpeixen aquestes formes en el món real, però resulten visualment força plausibles. Una bona referència on trobar exemples d'aquestes funcions i com construir-les és [Ebert02]. Aquest llibre, tot hi que tracta de generar patrons de forma procedural en general, també concreta en el tema dels terrenys.

Com hem dit, utilitzarem funcions de soroll fractal per a generar els camps d'altures que conformaran el terreny. Aquestes funcions és caracteritzen per ser formades per a diferents capes de soroll coherent a diferents escales. Típicament, la relació entre les diferents capes de soroll ve parametritzada per a diferents variables que permeten regular les característiques dels terrenys generats. Podem enumerar els paràmetres més bàsics de la següent forma:

- Octaves: El nombre de capes de soroll.
- Lacunaritat: L'escala planar entre les diferents capes.
- H: L'increment fractal.

A partir d'aquests paràmetres la funció de soroll fractal més senzilla que es pot descriure per un punt p és la següent:

Algorisme 3 Funció fBm clàssica

function FBM($p, octaves, lacunarity, H$)
  $v \leftarrow 0$
  $o \leftarrow 0$
  while $o < octaves$ **do**
    $v \leftarrow v + noise(p) * lacunarity^{-H*o}$
    $p \leftarrow p * lacunarity$
    $o \leftarrow o + 1$
  end while
  return v
end function

On $noise()$ és una funció de soroll coherent com les que hem vist a la secció anterior. Noteu que a cada iteració sumem una freqüència de soroll inferior escalada per el factor $lacunarity^{-H*o}$ que,

com es pot veure, serà més petit a mesura que passin les iteracions. El comportament d'aquesta funció el podem veure a la figura 2.4:

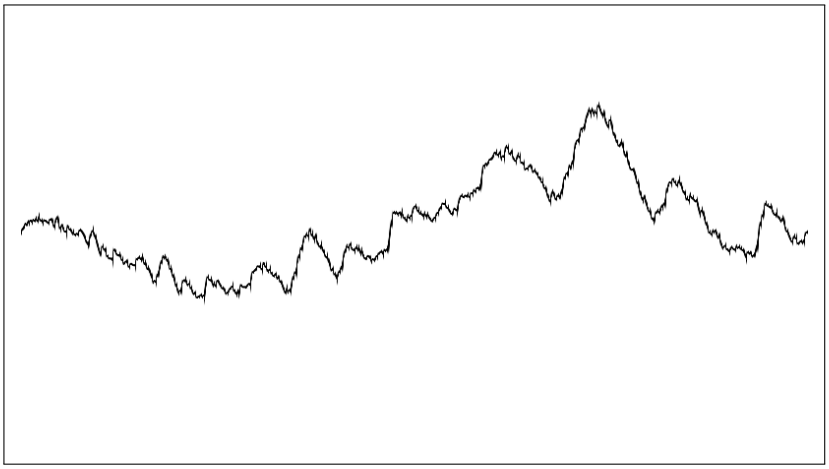


Figura 2.4: Una representació del perfil del tipus de soroll que s'aconsegueix amb la funció presentada a l'algorisme 3.

Els patrons generats per aquesta funció tenen les propietats de ser homogenis (estadísticament iguals a tot arreu) i isotròpics (estadísticament iguals en totes direccions). Aquestes propietats rarament es troben en la naturalesa, per tant, si volem introduir més realisme, haurem de fer alguna modificació a la funció. Això ens porta a parlar de les funcions multifractals. La característica principal de questes funcions és que els patrons generats són heterogenis, és a dir, no són estadísticament iguals a tot arreu. Una forma de descriure el comportament d'aquest fractals és dient que varien la seva dimensió depenent de la posició. La funció més simple d'aquest tipus, cosina de l'anterior és pot definir de la següent forma:

Algorisme 4 Funció fBm multifractal

function MULTIFBM($p, octaves, lacunarity, H, offset$)
  $v \leftarrow 1$
  $o \leftarrow 0$
  while $o < octaves$ **do**
    $v \leftarrow v * (noise(p) + offset) * lacunarity^{-H*o}$
    $p \leftarrow p * lacunarity$
    $o \leftarrow o + 1$
  end while
  return v
end function

Podem veure que la funció és extremadament similar a l'anterior. En aquest cas en comptes de sumar a cada iteració el valor de soroll, multiplicarem. També cal veure que hem introduït un nou paràmetre, $offset$. Aquest paràmetre determina la mulifractalalitat del resultat, és a dir, la variació depenent de l'escala i posició. A zero obtenim la màxima heterogeneïtat mentre que per valors grans el resultat és pla. Aquesta funció, com és de preveure, no és gaire estable. A mesura que augmenten les octaves tendirà a convergir cap a zero o infinit. A més, petites variacions en

el paràmetre de $offset$ poden fer que l'escala variï incontroladament. Per això deixarem aquesta funció com a model teòric. De fet, es fa molt difícil obtenir un resultat que es pugui mostrar de forma comparativa a la imatge de la funció anterior i per tant justifiquem així la falta d'aquesta.

Com que ens interessa el comportament de la primera funció però també voldríem obtenir certa multifractalitat com en la segona, una de les solucions que es planteja és un funció híbrida que mescli les propietats de les dos. Aquest tipus de funcions sovint són creades utilitzant la intuïció per a parametritzar certes propietats dels terrenys reals. En la següent, s'intenta modelar alguns efectes de l'erosió, en aquest cas, el fet de que els pics de les muntanyes són escarpats i les valls suaus.

Algorisme 5 Funció de fBm heterogènia

function HETEROFBM($p, octaves, lacunarity, H, offset$)
  $v \leftarrow noise(p) + offset$
  $p \leftarrow p * lacunarity$
  $o \leftarrow 1$
  while $o < octaves$ **do**
    $i \leftarrow (noise(p) + offset) * lacunarity^{-H*o}$
    $v \leftarrow v + v * i$
    $p \leftarrow p * lacunarity$
    $o \leftarrow o + 1$
  end while
  return v
end function

Com podem veure, per aconseguir aquest comportament es multiplica el valor de la octava a afegir per el valor que la funció tenia fins el moment. Així en zones on les primeres octaves hagin donat un valor baix, i que representen la vall, es reduirà molt notablement l'efecte de les successives iteracions. En canvi en zones altes, l'efecte es veurà reforçat. Noteu que la primera octava no està afectada de cap manera. En la figura 2.5 podem veure la classe d'estructures que es generen.

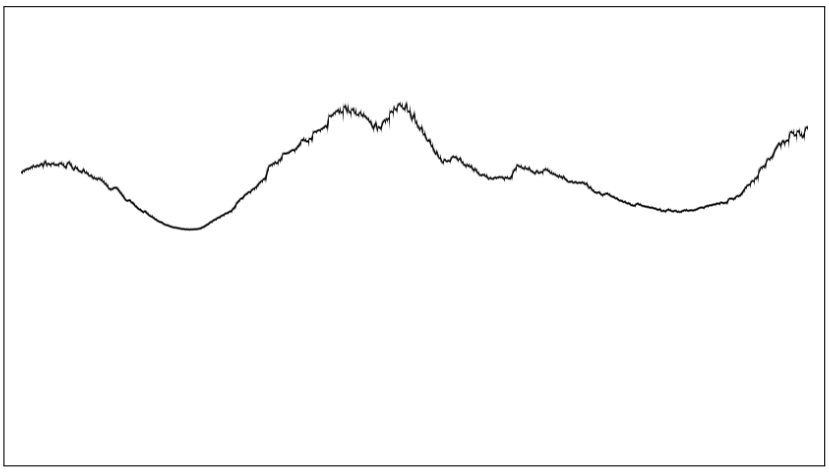


Figura 2.5: Una representació del perfil del tipus de soroll que s'aconsegueix amb la funció presentada a l'algorisme 5.

Aquesta última funció proporciona resultats visualment bons i té un comportament força controlable. És per això que l'hem escollida per a generar els nostres terrenys. Cal dir que hi ha multitud de altres funcions que també proporcionen resultats bons però de moment en el nostre projecte només suportem la creació de terreny amb una sola funció preestablerta. Deixem per a futures revisions la possibilitat de poder escollir diferents funcions de generació.

2.0.4 Obtenció del vector normal de la superfície

Sovint serà necessari no només conèixer l'altura en un punt sinó també la normal de la superfície. A vegades aquesta normal és pot aconseguir analíticament derivant les funcions de soroll. Tot hi que aquests mètodes acostumen a ser extremadament eficients, no són pràctics ja que cal implementar-ne una versió diferent per a cada funció de soroll. Sovint el que s'acaba utilitzant és una aproximació d'aquesta derivada utilitzant el mètode del punt mitjà. Aquest mètode el podem aplicar en un punt $p = (x, z)$ calculant els producte vectorial normalitzat entre els vectors següents:

$$\begin{aligned} \vec{v} &= (x - \varepsilon, f(x - \varepsilon, z), z) - (x + \varepsilon, f(x + \varepsilon, z), z) \\ \vec{u} &= (x, f(x, z - \varepsilon), z - \varepsilon) - (x, f(x, z + \varepsilon), z + \varepsilon) \end{aligned} \tag{2.13}$$

On $f(x, z)$ és la funció de mostreig de l'alçada de la superfície i ε dependrà de la resolució del mostreig.

Capítol 3

Gestió computacional del terreny

Un cop vista en la secció 2.0.1 la representació computacional dels terrenys que utilitzarem, cal buscar una forma eficient de visualitzar-los. Donada la popularitat dels terrenys representats com a camps d'alçades, en la literatura trobem força tècniques per a la visualització d'aquests. Com em vist, els nostres terrenys seran no-finits i per tant podem descartar totes aquelles tècniques que no suportin un actualitzat incremental de les dades dels camps d'altures. A més, hem dit que volíem generar totes les dades a la GPU per tal d'aprofitar el gran paral·lelisme dels algorismes generadors de terrenys i estalviar-nos transferir dades entre la GPU i CPU. La compatibilitat amb aquest esquema és imprescindible. Un altre aspecte que ha de solucionar la tècnica escollida és el filtrat dels camps d'alçades allí on el detall no sigui útil, sobretot les mostres llunyanes a la càmera. Això és important ja que volem visualitzar terrenys molt amplis i, si no reduíssim el nombre de mostres, podríem tenir problemes de rendiment.

En general, es poden trobar dos categories de tècniques que s'adapten als requisits que hem exposat:

1. Tècniques basades en la subdivisió de l'espai en forma de quadtree per tal d'acomodar el detall necessari.

2. Tècniques que es basen en mantenir graelles regulars a diferents resolucions.

Nosaltres ens centrarem en la segona categoria ja creiem que una implementació que compleixi amb els requisits que hem establert és més senzilla amb aquesta. Cal dir que els dos paradigmes comparteixen similituds i que les úniques raons de per les quals que hem escollit la segona són per la seva aparent senzillesa i la seva popularitat.

En la següent secció explicarem en detall la tècnica anomenada Geometrical Clipmaps que utilitzarem per a actualitzar i visualitzar el terreny.

3.1 Geometrical Clipmaps

En aquesta secció presentarem la tècnica escollida per a visualitzar el terreny generat. Aquesta, anomenada Geometrical Clipmaps, la trobem explicada en dos articles [Losasso04] i [Asirvatham05]. La primera cita és un primer treball on trobem una descripció detallada de la tècnica. En la segona és descriu una implementació completament basada en GPU de les idees del primer

amb notòries modificacions per tal d'adaptar-les a aquest paradigma. Nosaltres seguirem el segon article [Asirvatham05] ja que, com veurem més endavant, els plantejaments que s'hi descriuen s'adapten molt bé als nostres requisits.

A continuació farem una exposició detallada de les idees que es descriuen en l'article mencionat anteriorment tot indicant els canvis que hem introduït. També intentarem aclarir, amb la nostra interpretació, alguns punts de l'article que ens han semblat una mica vagues o que creiem que necessiten una explicació més detallada. Primer, en la subsecció 3.1.1, explicarem en que consisteix la tècnica. Després, en les següents subseccions (3.1.3 i 3.1.2) s'adreça amb detall els dos temes clau de la tècnica: la visualització del terreny i l'actualització de les dades que el defineixen.

3.1.1 Descripció General

Geometrical clipmaps és una tècnica de visualització en temps real de terrenys representats com a camps d'altures. Aquests camps d'altures venen definits per un conjunt de mida arbitraria de mostres distribuïdes en forma de graella 2 dimensional, quadricular i regular tal com s'han presentat el la secció 2.0.1. En general, no és possible tenir una representació completa d'aquests en memòria gràfica, per això es defineix un esquema de memòria cau que emmagatzema les dades del terreny locals a la càmera virtual i que s'actualitza de forma incremental a mesura que aquesta es mou. Aquesta memòria esta dividida en diferents nivells de detall, cada un representa el terreny a una escala superior. Concretament cada nivell és el doble de gran que l'anterior. Al mateix temps, cada nivell té la meitat de detall que l'anterior. Definim el nivell de detall com el nombre de mostres per unitat de superfície. Si assumim que les mostres dels diferents nivells estan alineades, podem veure que el nombre d'aquestes és exactament el mateix en tots els nivells de la memòria cau. Així podem reservar L porcions de memòria gràfica (essent L el nombre de nivells de detall) de $n \times n$ mostres. A la figura 3.1 podem veure una representació gràfica d'aquest esquema.

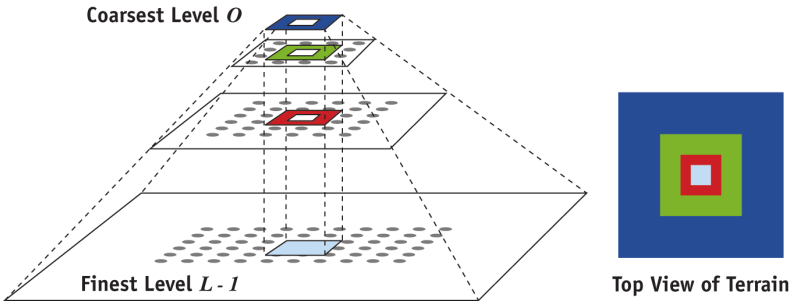


Figura 3.1: Cada nivell de detall té la mateixa petjada de memòria al mateix temps que cobreix el doble de superfície que l'anterior. Imatge extreta de [Asirvatham05]

Per motius d'eficiència i comoditat, com veurem a les seccions 3.1.3 i 3.1.2, en la nostra versió hem forçat que L i n siguin potencies de 2. Cal dir que el fet de que la porció d'espai que ocupa cada nivell sigui quadrada no és casualitat. En general, la rotació de la càmera pot set molt ràpida i impredictible, per tant és necessari tenir cobertes totes les direccions en tot moment. Com que la memòria té una disposició rectangular, la forma més regular que podem aconseguir sense malgastar-ne és la de quadrat per això s'utilitza aquesta forma.

Aquesta forma de organitzar les mostres redueix la memòria i el cost de visualitzar els camps

d'altures. Donada una superfície que volem representar amb $m \times m$ mostres (en el seu nivell de detall més elevat) utilitzant L nivells de detall els requeriments de memòria venen definits per la següent expressió:

$$\mathcal{O}L\left(\frac{m}{2^{L-1}}\right)^2 \tag{3.1}$$

El raonament al darrera de l'expressió és el següent: Sigui n l'arrel quadrada del nombre de mostres de un dels nivells. En el nivell amb més detall cobrim una superfície de n^2 per tant amb el nivell l cobrirem una superfície de $(2^{l-1}n)^2$ per la definició de l'escala dels nivells de detall. Substituint l per L trobem l'extensió que ocupa el nivell més gran i per tant la extensió total de del terreny emmagatzemat. Aquesta ha de ser igual a m^2 del qual s'en deriva que:

$$n = \frac{m}{2^{L-1}} \tag{3.2}$$

Aquí cal puntualitzar que n ha de ser natural i per tant no podrem contenir extensions arbitraries de terreny. Això no és problema ja que els camps d'altures que volem representar sempre seran més grans que l'extenció que cobreix la memòria cau. Seguint amb el raonament, com que hem vist que tots els nivells requereixen el mateix nombre de mostres, necessitarem espai per a L nivells de $n \times n$ mostres, d'aquí l'expressió 3.1. Aquest compromís entre memòria necessària i reducció del detall no suposa cap disminució en la qualitat percebuda. Això és degut a que, com veurem més endavant en la secció 3.1.3, el detall llunyà a la càmera no és útil.

Com hem dit abans, la memòria s'actualitza de forma que la càmera sempre estigui mes o menys centrada respecte el quadrat format per les $n \times n$ mostres de cada nivell. Típicament el moviment d'aquesta última és coherent i per tant la modificació de les mostres és incremental. Degut a la diferència d'escala de cada nivell la freqüència d'actualització de cada un d'aquests és diferent. Suposant que el màxim nivell de detall és la unitat, tenim L nivells i la mida d'aquests és de $n \times n$, el nombre de mostres a modificar en un desplaçament horitzontal de d unitats ve donat per la següent expressió:

$$n \sum_{i=0}^{L-1} \left\lfloor \frac{d}{2^i} \right\rfloor \tag{3.3}$$

Podem veure que:

$$\sum_{i=0}^{L-1} \left\lfloor \frac{d}{2^i} \right\rfloor \leq d \sum_{i=0}^{L-1} \frac{1}{2^i} = d\frac{2^L - 1}{2^{L-1}} < 2d \tag{3.4}$$

La part de la dreta de l'expressió anterior s'obté del fet que estem davant de una successió geomètrica. D'aquí és fàcil veure que hi ha un límit superior de $2dn$ en les mostres que s'han d'actualitzar. Anàlogament podríem fer el mateix raonament per un desplaçament vertical o un on intervinguin les dos components, tenint en compte que, en aquest últim cas, hi haurà una part de les mostres que són comunes a les dos direccions. També cal dir que en la majoria dels casos aquest resultat està molt lluny de el valor real. Sigui t el temps necessari per desplaçar la càmera d unitats horitzontalment, escollint un d suficient gran com per què s'actualitzin tots els nivells, és a dir, $\left\lfloor \frac{d}{2^{L-1}} \right\rfloor \geq n$ tenim que $d \geq 2^{L-1}n$. Normalment la velocitat de la càmera és inferior a n u/t (u és distancia i t és temps), per tant $t \geq 2^{L-1}$. Amb aquest raonament volem mostrar que el temps necessari per aproximar-nos al límit teòric que hem establer anteriorment normalment és gran i per tant el cost de actualitzar la memòria cau queda temporalment distribuït. En l'explicació anterior hem dit que la velocitat de la càmera normalment és inferior a n u/t, això és degut a que l'esquema de memòria té la propietat de modificar el nivell de detall general del

terreny activant o desactiva nivells segons convingui. Quan la velocitat de la càmera és tal que s'han d'actualitzar totes les mostres del nivell actiu amb més detall, no té sentit mantenir-lo actiu ja que no contribueix a la qualitat visual. Aquí cal afegir que perquè aquesta última explicació tingui sentit em de definir d'alguna manera la unitat de temps. Aquesta unitat serà el temps transcorregut entre actualitzacions de la imatge que representa el terreny. Un altre cas on és útil disminuir el detall del terreny és quan la posició del la càmera està molt per sobre de l'elevació del terreny. D'aquesta manera s'evita el sobre mostreig a l'hora de visualitzar la la porció del terreny en qüestió.

Per a visualitzar les mostres guardades s'utilitzen L graelles triangulades regulars de punts(o, com que estem parlant d'un espai topològic, vèrtexs) creant una malla per a cada nivell de detall. Les posicions dels vèrtexs tenen una correspondència exacta amb les mostres de cada nivell. D'aquesta forma modificant la component vertical d'aquests vèrtexs per l'elevació de les mostres a les que corresponen obtenim una superfície que representa el terreny. Podem veure que tal com estan definides aquestes graelles, la que cobreix més superfície conté totes les altres. Per això el nucli d'aquesta queda representat per els altres nivells amb més detall i no cal mostrar-lo. Així la graella no cal que sigui complerta si no que té un forat en el seu centre les dimensions del qual són la meitat de la mida que ocupa la graella creant una mena d'anell. Aquesta característica es propaga a tots els nivells excepte el de més detall que, òbviament, ha de cobrir tota la superfície. A les figures 3.2 i 3.1 podem visualitzar aquest comportament.

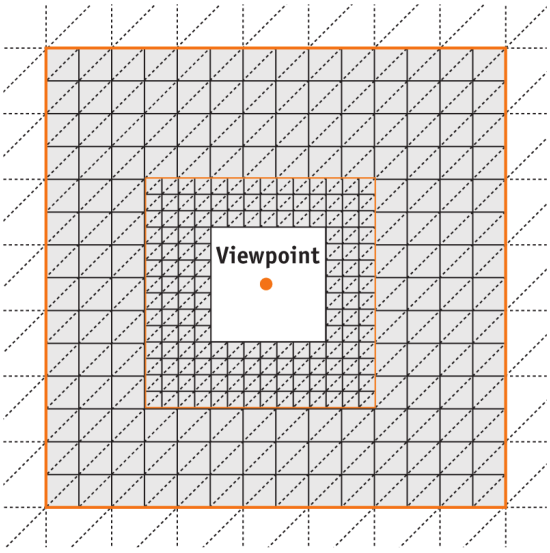


Figura 3.2: Anelles de geometria que conformen els diferents nivells de detall. Imatge extreta de [Asirvatham05].

Aquesta forma de procedir genera una frontera entre nivells de detall que en general no és tancada ja que hi ha forats deguts a les discrepàncies de detall entre diferents nivells. A més, aquestes fronteres són molt visibles, sobretot quan la càmera és mou, i per tant redueixen la qualitat visual percebuda del terreny representat. La tècnica soluciona aquest problema introduint una regió de transició suau entre nivells. D'aquesta forma el detall es va perdent de forma progressiva arribant a la frontera amb el mateix que el del següent nivell. Una altra forma de veure-ho és dient que la malla de més detall es transforma de forma suau en la de menys detall a mesura que els vèrtexs

de la primera estan més a prop de la frontera entre aquestes dos.

En les següents seccions discutirem amb detall la nostra implementació dels conceptes aquí mostrats utilitzant les tecnologies que em cregut oportú per a la realització d'aquest projecte (OpenGL i CUDA). Com hem dit abans, ens basarem en la implementació presentada a [Asirvatham05] la qual intenta utilitzar al màxim el *hardware* gràfic (GPU) reduint a mínims la comunicació entre aquest i el *hardware* de propòsit general (CPU). A més, nosaltres en comptes de utilitzar camps d'altures emmagatzemats en una base de dades com fa l'article, generarem les mostres directament en memòria de vídeo (de GPU) així estalviant-nos transferències de dades entre CPU i GPU. Donada aquesta diferència, ometrem tota la part de l'article dedicada a la síntesi i compressió del detall.

3.1.2 Memòria cau

En aquesta secció descriurem detalladament la implementació de la memòria cau on mantindrem les mostres locals a la càmera. Aquesta memòria ha de ser accessible per els dos marcs amb els que treballarem. Concretament ha de poder ser llegida per a OpenGL com a una textura i ha de poder ser escrita i llegida per a CUDA. La opció més directa, i la que hem implementat, és reservar amb OpenGL la memòria necessària i després, a través de la API (*Application Programming Interface*) de inter operabiliat que proporciona CUDA, registrar aquesta memòria en objectes CUDA de `Surface`. Aquest objectes, a diferència dels de `Texture` no només permeten llegir, si no que també permeten escriure. Cal notar que no és possible reservar amb CUDA i registrar a OpenGL. Com hem vist abans, tots els nivells de detall ocupen el mateix espai, per tant podem utilitzar l'estructura de textura de OpenGL anomenada Texture 2D Array. Aquesta estructura és una col·lecció ordenada de textures 2 dimensionals que tenen les mateixes dimensions. Malauradament, CUDA no suporta, a dia d'avui, la interoperació directa d'aquest objectes de OpenGL. El que si que podem fer però, és registrar element a element com si fossin textures separades. Estranyament existeix un objecte amb les mateixes característiques que els Texture 2D Arrays de OpenGL a CUDA però sembla que no existeix forma de registrar l'un a l'altre. Aquests conjunt de textures només cal reservar-les i registrar-les un cop a l'inici de forma que el cost associat a aquestes operacions, que no és menystenible, no té cap mena de repercussió en el rendiment de les operacions de actualització o visualització. Un altre punt a tenir en compte és el format d'aquestes textures. Com que voldrem guardar tant les altures com les normals de la superfície que representa el terreny, utilitzarem una format de 4 elements (3 per la normal i 1 per l'alçada). Per evitar qualsevol tipus de problema de precisió utilitzarem nombres en punt flotant de 32 bits, en OpenGL `GL_RGBA32F`.

Un cop vista l'organització de la memòria passarem a examinar l'actualització d'aquesta. Quan la càmera és mogui lateralment, per tal de que les mostres que conté la memòria cau estiguin centrades en aquesta, caldrà substituir algunes d'aquestes per a noves dades. Sigui $\vec{\delta} = (\delta_x, \delta_y)$ la projecció en el pla horitzontal del vector de translació experimentat per la càmera. En general, l'àrea de mostres que cal incorporar a la memòria és pot descriure amb dos rectangles la suma de l'àrea dels quals té forma de L majúscula com es mostra a la figura 3.3. Aquests dos rectangles els diferenciarem segons si la orientació del seu costat més llarg és vertical (V) o horitzontal (H) i els podem definir amb un punt (v i h respectivament en la figura 3.4) que representen la cantonada inferior esquerra del rectangle i les seves dimensions. Per a facilitar el posicionament d'aquests

punts fixarem l'origen de coordenades a la part inferior esquerra de l'extensió que ocupa el nivell la memòria cau que estiguem tractant abans de l'actualització (o en la figura 3.3).

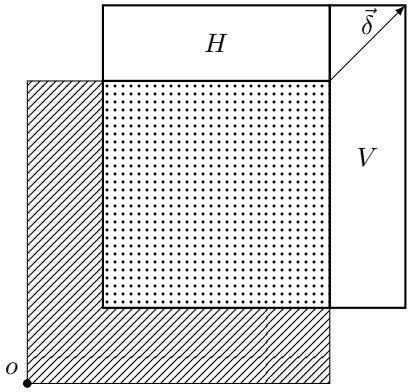


Figura 3.3: Desplaçament de δ unitats de la memòria cau. La regió rallada mostra la zona que és expulsada, la regió puntejada la zona mantinguda i la regió llisa, dividida en dos parts, la zona incorporada.

A continuació descriurem la posició d'aquest punts i les dimensions dels rectangles per el nivell amb més detall. Aquests paràmetres es descriuen anàlogament, escalant el vector $\vec{\delta}$, per els altres nivells de detall.

Una forma compacta de obtenir les coordenades dels punts anteriorment mencionats podria ser la següent:

$$\begin{aligned} v_x &= (1 - p(\delta_x))\delta_x + p(\delta_x)n \\ v_y &= \delta_y \end{aligned} \tag{3.5}$$

i

$$\begin{aligned} h_x &= p(\delta_x)\delta_x \\ h_y &= (1 - p(\delta_y))\delta_y + p(\delta_y)n \end{aligned} \tag{3.6}$$

on $p(x)$ és defineix de la següent forma:

$$p(x) = \begin{cases} 1, & \text{si } x > 0 \\ 0, & \text{altrament} \end{cases} \tag{3.7}$$

Es pot veure que les expressions 3.5 i 3.6 generen correctament una de les 4 úniques configuracions possible mostrades a la figura 3.4 depenent del signes de $\vec{\delta}$.

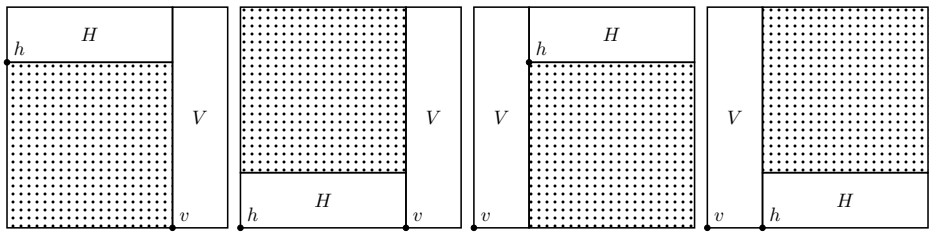


Figura 3.4: Representació de les 4 configuracions possibles en l'actualització de la memòria cau. Seguim amb el mateix esquema de colors que en la figura 3.3

Les dimensions dels rectangles es defineixen de la següent forma:

$$\begin{aligned} V_w &= |\delta_x| \\ V_h &= n \end{aligned} \tag{3.8}$$

i

$$\begin{aligned} H_w &= n - |\delta_x| \\ H_h &= |\delta_y| \end{aligned} \tag{3.9}$$

on el subíndex w i h indiquen amplada i alçada respectivament. També cal que recordem que n és la mida de costat de un nivell de la memòria cau i $n < \delta_x$.

Aquesta forma de descriure les regions proporciona una visió clara de com dividir la tasca per aprofitar el model de còmput paral·lel de CUDA a l'hora de generar el camp d'altures. A continuació farem una petita explicació dels conceptes d'aquest model que ens interessen, concretament els que ens permeten distribuir la tasca entre els recursos computacionals que ens brinda el hardware especialitzat que volem utilitzar i que podem trobar explicats a [NVIDIA17]. No entrarem en detall en els models de memòria ja que, com veurem, no tindran una afectació notable en el resultat de les operacions.

En aquest model es defineixen 3 nivells de granularitat lògica en els que es pot dividir una tasca. Aquesta granuralitat és jeràrquica, de més general a més concret, tenim el nivell de grid, el de block i el de thread. Les tasques es defineixen per mitjà de kernels que seran executats per una grid, és a dir, un conjunt de threads organitzats en blocks. Els blocks de una grid són independents els uns dels altres mentre que els threads de un block poden comunicar-se entre ells a través de la memòria que comparteixen. En la figura 3.5 podem veure una representació gràfica d'aquesta jerarquia.

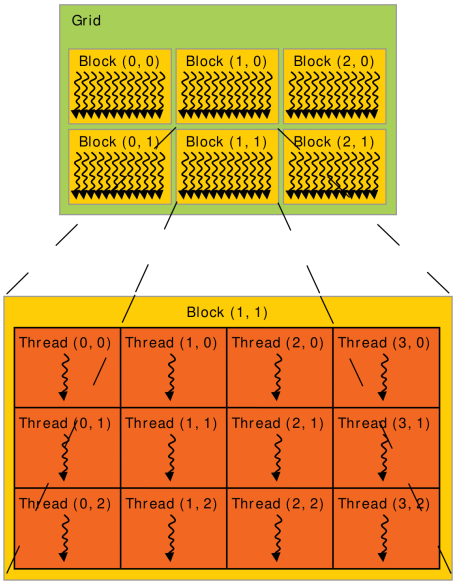


Figura 3.5: Organització jeràrquica dels threads de CUDA. Imatge extreta de [NVIDIA17].

Així doncs, en el nostre cas, utilitzarem dos kerenels, un per cada àrea (H i V). Podem veure que la tasca serà la mateixa per les dos simplement modificant els paràmetres que defineixen els

rectangles que hem definit abans. Aquí un podria pensar que si la tasca és la mateixa per a les dos àrees podria ser bona idea unificar-les, però, com veurem a continuació, això complicaria la distribució del treball en blocks de threads. De fet, dividir el problema en dos parts no és casual, si no que hi ha una intenció clara per facilitar la distribució del treball. Quan executem un kernel haurem d'indicar en quants blocks i quants threads per block volem que s'executi. Multiplicant aquests dos valors obtindrem la concurrència o nombre de fils d'execució paral·lels que utilitzarà aquests kernel. Aquest blocks i threads poden ser especificats en 1, 2 o 3 dimensions. Donada la naturalesa 2 dimensional del nostre problema utilitzarem aquesta modalitat tant per designar blocks com per designar threads. Existeixen límits sobre el nombre de blocks actius en un cert instant, per tant, escollir un block massa petit pot resultar que alguns dels threads no puguin executar-se. A més, també existeix un límit en la quantitat de threads que pot contenir un block, típicament, en el hardware actual, 1024. Un nombre que es sol recomanar és 256 threads per block. Aquest nombre garanteix que tots els recursos de còmput seran utilitzat (màxima ocupació) al mateix temps que no supera els límits establerts per el hardware. Una forma fàcil d'organitzar 2 dimensionalment 256 threads és en blocks de 16×16 threads. D'aquesta forma podem computar la grid de blocks de la següent forma:

$$\begin{aligned} B_x &= \lceil (R_w/16) \rceil \\ B_y &= \lceil (R_h/16) \rceil \end{aligned} \tag{3.10}$$

On B_x i B_y són respectivament el nombre de blocks en les coordenades x i y de la grid i R_w i R_h són les mides (amplada i alçada respectivament) de la porció rectangular que volem computar. Noteu que així a cada thread se li assigna una mostra diferent. Aquests blocks no necessàriament cal que siguin quadrats però, com veurem a continuació, aquesta característica facilita la distribució del treball independentment de les dimensions de R. Com podem veure, si les mides de R no són múltiples de 16, tindrem més threads que mostres volem generar, per tant, serà imprescindible comprovar si la mostra que se li ha assignat al thread es troba inclosa en la regió que volem computar. La feina assignada a un block (de 256 threads) es computa en 8 wraps (o grups) de 32 threads físics, aquest nombre (el nombre de therads per wrap) ve fixat per CUDA i es el conjunt mínim de threads que s'executaran concurrentment. Això implica que si en el conjunt d'aquest 32 threads hi ha 1 element que s'ha de computar i 31 que no, tindrem 31 threads esperant que aquest únic acabi la feina. Com és evident, intentar minimitzar el nombre de threads que esperen sense fer res és clau a l'hora d'aconseguir la màxima eficiència. Distingirem tres tipus de wraps segons els diferents escenaris que es poden donar:

1. Totalment utilitzat: A tots els threads se'ls assigna computar mostres de dins de la regió.
2. Totalment no utilitzat: A tots els threads se'ls assigna computar mostres fora de la regió.
3. Parcialment utilitzats: Alguns threads se'ls assigna computar mostres de dins de la regió i altres de fora.

Els wraps de tipus 1 i 2 direm que són òptims: O tots els threads fan feina (tipus 1) o tots els threads alliberen els recursos computacionals ràpidament (tipus 2). El de tipus 3 en canvi desaprofita alguns d'aquest recursos. Podem veure que el nombre de de wraps de tipus 3 que obtindrem amb l'organització que hem descrit abans és funció de com el threads de block s'assignen als threads de wrap i de R (dimensions de la regió de mostres a computar). Aquesta assignació, que sempre és la mateixa, es fa en ordre ascendent de identificador de thread de block. Aquest

identificador és computat, en el cas del thread (x, y) de un blocks 2d de dimensions (D_x, D_y), per l'expressió:

$$id = x + yDx \tag{3.11}$$

Amb aquesta forma de enumerar els threads de wrap podem veure que cada wrap serà format per dos files de threads del block. Si calculem la posició que cada thread a de calcular per mitjà d'aquest identificador de thread de block, és fàcil veure que obtindrem pocs wraps de tipus 3 si $R_x > R_y$ en canvi si, pel contrari, $R_x < R_y$ moltes d'aquestes files quedaran partides deixant una quantitat notable de threads de wrap inactius. Per corregir aquest comportament només cal interpretar la posició a computar de forma diferent segons si $R_x < R_y$ o $R_x > R_y$. D'aquesta forma podem calcularem les coordenades de l'àrea de mostres que li toquen a cada thread de la següent forma:

$$\begin{aligned} x &= \begin{cases} R_x > R_y, & \texttt{threadIdx.x + blockIdx.x * blockDim.x} \\ altrament, & \texttt{threadIdx.y + blockIdx.x * blockDim.y} \end{cases} \\ y &= \begin{cases} R_x > R_y, & \texttt{threadIdx.y + blockIdx.y * blockDim.y} \\ altrament, & \texttt{threadIdx.x + blockIdx.y * blockDim.x} \end{cases} \end{aligned} \tag{3.12}$$

Aquí hem utilitzat les variables que en proporciona CUDA que identifiquen el thread, `threadIdx.x` i `threadIdx.y` són les components x i y respectivament de l'identificador del thread en el block, `blockIdx.x` i `blockIdx.y` són les components x i y respectivament de l'identificador del block i `blockDim.x` i `blockDim.y` les dimensions del block. L'únic que em fet és intercanviar x per y del thread si $R_x > R_y$. Cal notar que no intercanviem les coordenades de block. En el nostre problema sovint haurem de computar regions molt allargades, per això fer aquest tipus d'observacions és important si volem aprofitar al màxim el hardware del que disposem. Una altra propietat del nostre problema és que, com hem vist en la secció de generació, totes les mostres es poden computar independentment de les altres. No hi ha dependències de cap tipus. Això simplifica enormement la tasca ja que, per una banda, no cal que pensem cap tipus de sincronització entre threads per tal d'evitar condicions de carrera (l'ordre d'execució no importa) i per l'altra, no cal que recuperem cap resultat anterior guardat a memòria, per tant, també podem estalviar-nos inquirir el model de memòria que presenta CUDA.

Un cop vist com calcular les coordenades de les mostres de una d'aquestes regions, calcular la coordenada real del nostre camp d'altures és senzill si coneixem l'origen de la regió en l'espai que defineix el camp d'altures a un determinat nivell de detall. Noteu que la unitat és diferent a cada nivell de detall. Aquesta coordenada és precisament la que hem calculat anteriorment i que hem anomenat h i v respectivament per cada una de les regions H i V. Com que per calcular el valor de la mostra necessitem les coordenades descrites respecte el nivell amb detall més alt (nivell 0), caldrà multiplicar les coordenades per 2^l on l és el nivell de detall.

Un cop calculat el valor de la mostra utilitzarem un mapejat toroïdal sobre les coordenades en espai del nivell, per trobar la coordenada de la `Surface` on guardarem la mostra. Aquest mapejat ens proporciona una forma eficient de representar incrementalment una superfície no acotada i es defineix la següent manera:

$$x' = x \pmod{n} \tag{3.13}$$

On x és la coordenada a transformar, x' la coordenada transformada i n és la mida de la regió

limitant. Aquí cal puntualitzar que si n és potència de 2, Aquesta operació és fàcil d'optimitzar com a:

$$x' = x\&(n-1) \tag{3.14}$$

On l'operador & indica la `AND` bit a bit de la representació en complement a dos de dos nombres enters. Aquesta optimització és útil ja que les operacions modulars amb enters en el hardware gràfic actual poden ser relativament costoses. És per això que forçarem que totes les textures utilitzades per guardar el camp d'altures siguin potència de 2.

Un tema que no hem tractat encara és la generació de les normals. Per a computar-les utilitzarem la mateixa divisió del treball que per les mostres de les alçades. Cal notar que existeix una dependència clara entre els dos càlculs. Tal com hem vist en la secció 2.0.4, les normals de un punt s'obtenen a partir de les alçades veïnes. Per tant és imprescindible que el kernel d'actualització de les alçades hagi acabat per tal de que el resultat sigui correcte. En el nostre cas això està garantit ja que només utilitzem un stream o context d'execució i per tant els kernels s'executen de forma seqüencial. Si volguéssim utilitzar més de un stream, per exemple per a computar les dos regions que hem descrit abans de forma concurrent, caldria sincronitzar-los abans de poder computar les normals. Una observació que podem fer és que les lectures de les alçades veïnes es faran utilitzant una `Surface`. Aquest tipus de memòria té polítiques de memòria cau basades en la localitat 2D la qual cosa afavoreix al rendiment amb el patró d'accés que utilitzem.

3.1.3 Visualització del terreny

A l'hora de visualitzar el terreny intentarem reproduir el més acurat possible les dades residents en la memòria cau que el defineixen en el moment de mostrar-lo. D'aquesta manera crearem graelles 2d triangulades els vèrtexs dels quals corresponen un a un amb les posicions del terreny guardades a la memòria cau. Com hem vist abans, la memòria està dividida en nivells de detall, per tant, haurem de ser capaços de reproduir tantes graelles com nivells. Podem veure que les graelles de cada nivell són exactament iguals sota escalat i translació. És per això que podem aprofitar els recursos que les defineixen instanciant múltiples vegades la mateixa malla. A més, podem notar que aquestes graelles són estructures extremadament regulars i que, per tant, les podem subdividir en porcions més petites repetibles sota translació per tal de estalviar encara més memòria gràfica. Aquesta subdivisió no només aporta una reducció de les dades que es necessiten per a descriure les graelles si no que proporcionen un marc per a descartar les porcions que no són visibles utilitzant tècniques de frustum culling així reduint el treball que haurà de realitzar la GPU.

Una observació que es pot fer és que si volem que el perímetre exterior de una anella interior encaixi en el perímetre interior de la del següent nivell amb menys detall el nombre de vèrtexs de costat n de l'anella ha de ser imparell. Com hem vist en la secció 3.1.2, les textures que contindran les dades del terreny seran potències de 2, per tant, per tal de aprofitar al màxim la textura fixarem $n = 2^k - 1$. A més, aquesta mida força que l'anella de més resolució estigui desplaçada una unitat respecte la de menys detall. Aquesta propietat és imprescindible ja que el desplaçament de la primera serà el doble de freqüent que el de la segona.

Utilitzarem la divisió del les anelles que es planeja a [Asirvatham05] de la qual en podem veure un exemple a la figura 3.6. Principalment es divideix cada anella en 3 tipus de primitives: blocks, fix-ups i trims. Els blocks són la part principal de l'anella. En total, per a descriure una anella,

utilitzarem 12 subgraelles de $m \times m$ vèrtex tals que $m = \frac{n+1}{4}$ disposats tal com es mostra a la figura 3.6 en gris. Aquests blocks no cobreixen la totalitat de l'anella i per tant introduirem les parts que falten amb una altra primitiva, els fix-ups (en verd a la figura 3.6). Aquesta estructura cobreix els forats que queden entre blocks i la seva mida serà de $m \times 3$. Donada la poca superfície que aquests elements representen es decideix mantenir els 4 que componen l'anella com una sola entitat. Per últim, tenim els trims, en blau a la figura 3.6. Aquest elements són els encarregats d'acomodar el desencaix entre nivells de detall. La posició d'aquest element dependrà de la posició relativa entre els dos nivells. Per entendre el moviment d'aquest element cal veure que l'anella amb més detall es mourà amb desplaçaments múltiples de l'escala de l'anella amb menys detall.

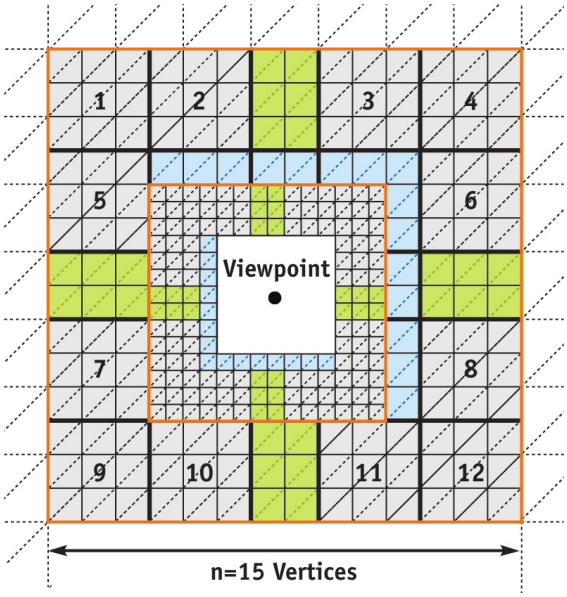


Figura 3.6: Divisió del l'anella de geometria per una graella de 15 vèrtexs de costat. Imatge extreta de [Asirvatham05].

Representació, Generació i Optimització de la geometria

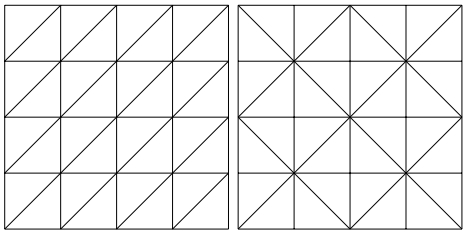
En general, la forma de representar qualsevol objecte geomètric per a ser utilitzat en el context de un motor de rasteritzat és el de una col·lecció de triangles. Aquest conjunt, en la forma més bàsica, consisteix en una llista de triplets de vèrtexs representats per les seves coordenades en un ordre conegut. Per representar computacionalment les subgraelles que conformen les anelles de les quals que hem parlat anteriorment utilitzarem una llista de triangles indexada. Aquesta llista té dos components, per una banda una col·lecció ordenada de vèrtexs 2D i per l'altra una llista de triangles representats com a triplets d'índexs a la col·lecció de vèrtexs. Aquesta no és la única representació. Una altra possibilitat seria que aquests índexs representessin un *strip*, és a dir, una cadena de índex on cada un d'ells forma un triangle nou amb el dos elements anteriors. És evident que aquest tipus d'estructures redueix notablement el nombre d'índexs a canvi d'introduir certes complicacions i perdre flexibilitat a l'hora de descriure malles. Amb el hardware d'avui en dia (gran ample de banda, extensa memòria de vídeo, etc) els avantatges dels *strips* queden difuminats i per tant la flexibilitat que proporciona una llista de triangles pesa més a l'hora de escollir la representació de la malla.

Un cop vista l'estructura bàsica en la en la que descriurem la malla de la graella, discutirem sobre com generar-ne els triangles. En primer lloc veurem quin ordre utilitzar en la creació de la col·lecció de vèrtexs. Cal pensar que aquesta col·lecció haurà de ser indexada a l'hora de descriure els triangles i per tant una disposició que faciliti aquesta tasca és benvinguda. També cal veure que l'estructura final que tindrà aquesta col·lecció és lineal. Una forma de linealiar una graella 2d regular com la nostra és per files, és a dir, l'índex a l'estructura de un vèrtex arbitrari amb coordenades x i y és pot descriure de la següent forma:

$$i = y * w + x \tag{3.15}$$

On i és l'índex i w és el nombre d'elements de una fila de la graella. Cal dir que tan x com y són enters i que la distancia entre components dels elements és unitària, altrament aquesta transformació no funcionaria. Aquesta solució no és única, per exemple, un podria fer una linearització per columnes utilitzant un raonament similar o inclús es podria utilitzar una transformació més exòtica com podria ser el descrit per una corba de Hilbert. La simplicitat i la familiaritat amb la transformació descrita abans (per files) fa que sigui la utilitzada. Un cop vist com indexar els vèrtexs, podem procedir a discutir la generació de la triangulació que formarà la malla. Primer de tot en centrarem en veure les diferents formes que tenim de descriure una cel·la amb dos triangles, després discutirem si la disposició d'aquest triangles pot influir en el rendiment a l'hora de rasteritzar-los.

Per descriure les cel·les de la graella amb dos triangles cal que tinguem en compte principalment dos aspectes. Un d'ells es determinar quines combinacions dels 4 vèrtexs formen els triangles i l'altre l'ordre en el que aquests vèrtex són definits. Aquest ordre determina dos característiques dels triangles. Per una banda la orientació i per l'altra el *provoking* vèrtex. Per determinar la orientació utilitzarem la convenció antihorària, és a dir, el triangle estarà orientat cap en fora si l'ordre dels vèrtexs respecte el centre del triangle que el formen és antihorari i cap dins altrament. Aquí no tenim un sòlid tancat si una porció de un pla que s'estén indefinidament, per tant, una forma més clara de identificar aquestes orientacions seria com la part de dalt i la de baix. Aquesta orientació és important ja que el rasteritzador descartà, a mode de optimització, els triangles que, després d'aplicar les transformacions pertinents, mostrin la part interior de la geometria. Pel *provoking* vèrtex utilitzarem el conveni de l'últim vèrtex, és a dir, en el cas de que no s'interpolin els valors del tres vèrtex per generar les variables de sortida cap els fragments generats pel triangle (*flat shading*), el valor d'aquestes variables passa a ser el que en resulta d'aquest últim vèrtex. Aquest vèrtex, com veurem el la secció 4.0.1, ens interessa que sigui el mateix per els dos triangles que formen les cel·les i serà el que determini el subconjunt de vèrtexs que formen cada triangle. Arbitràriament fixarem que aquest vèrtex sigui el que forma la cantonada inferior esquerra de la cel·la. Cal notar que totes les cel·les presentaran la mateixa estructura (veure figura 3.7a) i per tant evitarem patrons en forma de creu que canvien la diagonal (o frontera entre triangles) de la cel·la depenent de la paritat d'aquesta. Un exemple d'aquest tipus de patrons el podem veure a la figura 3.7b.



(a) Patró regular (b) Patró altern

Figura 3.7: Diferents patrons de la malla

En la figura 3.8 es mostra la única forma de construir una cel·la que compleixi tots els requisits que hem fixat anteriorment. Podem veure que els triangles són definits pel triplet ordenat {3, 2, 0} i {1, 3, 0} i per tant compleixen tan les restriccions d'orientació (cap en fora) com de *provoking* vèrtex (l'últim vèrtex dels dos triangles és la cantonada inferior esquerra).

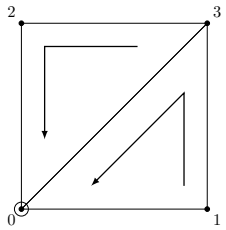


Figura 3.8: Construcció de una cel·la de la graella. La numeració dels vèrtex és a mode de etiquetatge, les fletxes indiquen l'ordre en que es construeixen els triangles i el *provoking* vèrtex s'indica amb un cercle.

Un cop vist com generar una cel·la de la graella, generar la resta pot semblar trivial. Malauradament això no és així. Una observació que podem fer és el fet de que cada vèrtex de la graella, exceptuant els que resideixen a l'exterior, és utilitzat per descriure 6 triangles (això es pot veure en la figura 3.7a). En principi, per a rasteritzar aquests triangles és necessari transformar tots els seus vèrtex, la qual cosa implica que la majoria de vèrtex es processaran 6 vegades. Aquest és un problema recurrent en qualsevol tipus de malla i per això existeixen optimitzacions de *hardware* que intenten disminuir aquest reprocessat. Una d'aquestes optimitzacions típicament consisteixen en mantenir una llista relativament petita que conté els últims vèrtex transformats. D'aquesta forma abans de processar el vèrtex es mira si aquest està en aquesta petita memòria cau (o *post-transform cache*). En cas afirmatiu s'utilitza el resultat guardat, altrament es processa el vèrtex i s'afegeix a la llista, possiblement fent fora un dels vèrtex residents. La política de reemplaçament que s'acostuma a utilitzar és el primer que entra és el primer que surt més coneguda com a *First-In-First-Out* o FIFO. Aquesta elecció es recolza en els resultats obtinguts en l'article [Hoppe99]. Com podem notar, l'efectivitat d'aquest tipus d'optimitzacions ve molt determinada per l'ordre en que aquests triangles siguin processats, per tant, és interessant tenir-ho en compte a l'hora de generar la graella.

Existeixen diversos algorismes que donada una malla arbitraria intenten reconfigurar-la per tal de que aprofiti millor aquests tipus d'esquemes. Un dels més nombrats, per la seva simplicitat, eficiència i generalitat, és el descrit a [Forsyth06]. Tot hi que segurament aquest tipus d'algorismes

millorarien el rendiment, la nostra malla és extremadament regular i té una forma molt concreta, per tant, és possible que hi hagin formes d'explotar aquesta optimització de forma més efectiva i controlada. Efectivament, a [Castaño09] trobem exactament això. L'article fa la observació de que si es processen el triangle per files de baix a dalt, els únics vèrtexs que es reutilitzen són els de la part de baix del triangle entrant, per tant, voldrem forçar que aquests vèrtexs ja siguin a la memòria cau quan processem un triangle. Si no fem res, l'ordre en que els vèrtexs entraran a la *cache* serà alternant la primera i la segona fila de vèrtexs de la graella. Això te el problema de que si la mida de la *cache* no és prou gran com per encabir dos files de vèrtexs completes de la graella, arribarà un punt en el que traurem elements que formen part de la part de baix de triangles que arribaran més tard. La solució que es proposa consisteix en omplir la *cache* amb la primera fila utilitzant triangles degenerats (sense àrea). D'aquesta forma els primers vèrtex que sortiran sempre seran els de la primera fila. Recordem que la política de reemplaçament que s'utilitza és FIFO. Sigui w la mida de les files de la malla, podem observar que la mida de la *cache* ha de ser al menys $w+2$ per tal d'evitar processar algun vèrtex múltiples vegades. Com hem vist abans aquest tipus de *caches* solen ser petites, per tant, per aprofitar aquest esquema haurem de subdividir la malla en porcions de una mida que aprofiti les observacions anteriors. Noteu que aquí només parlem de l'amplada ja que l'algorisme no presenta cap restricció en l'alçada de la malla. També cal veure que s'ha de poder tenir una estimació a la baixa de la mida d'aquesta memòria abans de generar la graella. En general, com hem vist abans, una bona estimació d'aquest valor és 20 entrades.

A la pràctica aquesta optimització no ha resultat en un guany en el rendiment. Tot hi que no podem descartar un error en la nostra implementació, una de les explicacions més plausibles és el fet de que aquest tipus d'optimitzacions no sembla que s'adaptin gaire bé a les arquitectures de GPU unificades modernes com les descrites a [Kubisch15]. A aquest tipus d'arquitectura és altament paral·lela i per tant, el més probable és que no processi els vèrtex de forma lineal com s'assumeix en l'algorisme anterior. Cal dir que aquí estem fent conjectures i que aquesta observació no s'ha tingut en compte fins ben entrat en el desenvolupament d'aquesta part del projecte i per tant un exhaustiu estudi queda perdent per a futures revisions. A [Giesen11] trobem més detalls sobre els problemes que comportaria mantenir una FIFO de vèrtex processats en aquestes GPUs modernes i planteja possibles solucions. L'autor recalca que ell també està fent conjectures però que té motius sòlids per pensar que el que explica és, dins de un cert dubte, correcte. A continuació farem una petita descripció del model de *cache* que es planteja a l'article i després donarem alguna idea de com aprofitar-la per millorar el rendiment a l'hora de visualitzar la nostra graella regular sense entrar massa en detall. El model que planteja consisteix en dividir el conjunt de vèrtexs en lots (*batches*) de la mida del *wrap* (o grup de theads físics concurrents com els descrits a la secció 3.1.2) de forma seqüencial. Abans d'incloure el vèrtex en un lot és mira si aquest ha estat introduït anteriorment en aquest lot. En cas afirmatiu no cal fer res, en cas negatiu s'afegeix al lot. Quan el lot és ple, s'envia a processar i acte seguit es comença a omplir un altre lot. Aquest nou lot és totalment independent i no té en compte si algun dels vèrtex que se li assignin ja ha estat processat en un altre lot. Aquest procés es repeteix fins que s'han tractat totes els vèrtexs que conformen la llista de triangles. Cal notar que crear els lots és molt més ràpid que computar-los i, per tant, n'hi haurà múltiples processant-se en paral·lel. Assumint aquesta divisió per lots, una forma de reduir el reprocessament de vèrtex en una graella regular, per una mida de *wrap* de 32, és subdividir la malla en rectangles de 4×8 vèrtexs. Segons el que hem descrit anteriorment, aquests rectangles ocuparan completament el *wrap*. D'aquesta manera podem dir que el que estem fent és preestablir els lots dels que parlàvem de una forma més òptima.

És fàcil veure que una subdivisió candidata a ser òptima és aquella amb menys perímetre ja que seran els vèrtex de la frontera els que es transformaran múltiples vegades, tantes com regions veïnes tinguin. En el cas de subdivisions rectangulars, les cantonades seran processades 4 vegades mentre que la resta del perímetre ho seran 2. D'aquí concloem que per conjunts de 32 vèrtex la forma més òptima ha de ser en rectangles de 4×8 o alternativament de 8×4. La anàlisis que s'ha fet d'aquesta forma de organitzar la malla és molt crua i per tant, com hem dit abans, un estudi més exhaustiu de la tècnica queda pendent per a futures revisions.

Frustum Culling

Un dels problemes més recurrents en el món dels gràfics per computador és el de determinar si una porció de l'escena és visible o no. Típicament, la importància d'aquesta operació rau en el fet que no interessa utilitzar recursos computacionals per a mostrar elements que queden fora del camp de visió definit per la càmera virtual. Com hem vist anteriorment, la nostra escena esta formada per anells concèntrics de geometria centrats a la càmera, per tant, suposant un angle de visió horitzontal realista (entre 100 i 60 graus) gran part d'aquesta geometria no serà visible. Existeix una gran varietat d'algorismes que solucionen eficientment el problema de visibilitat. La majoria tenen en comú que utilitzen una representació simplificada de l'escena. Normalment els objectes d'aquesta es substitueixen per volums fàcils de parametritzar com poden ser esferes, cilindres o caixes que estimen a l'alça el volum real de l'objecte a provar. En el nostre cas tenim porcions de terreny, el blocks que hem vist abans, amb una petjada quadrada i per tant és clar que la millor opció aquí és la de les caixes. A més, aquestes petjades sempre estan alineades amb els eixos de coordenades la qual cosa simplifica encara més la representació d'aquest element de volum. Cal notar que no aplicarem aquest test a les parts de les anelles que no siguin blocks. Aquestes parts tenen una àrea molt petita i per tant no contribueixen en una millora del rendiment al mateix temps que compliquen lleugerament el test.

Un problema que podem trobar és el de determinar l'extensió vertical per a cada caixa. Aquesta dimensió hauria de venir determinada per el mínim i el màxim de les mostres contingudes en la porció corresponent. Obtenir aquests valors pot ser complicat i costós. Una solució simple i efectiva és la de utilitzar el màxim i mínim global ja que aquest valors, en general, són fàcils d'aconseguir o d'estimar a l'alça. Un cop vista la aproximació que farem de l'escenari, passarem a discutir l'algorisme que utilitzarem per determinar la visibilitat d'aquest. Donada la simplicitat de la nostra escena intentarem allunyar-nos de qualsevol algorisme que requereixi estructures de dades complexes. Aquestes estructures, tot hi ser molt útils per a col·leccions de objectes grans ja que descarten grans porcions d'espai amb poques instruccions, per a col·leccions computacionalment petites, com la nostra, no suposen cap avantatge respecte solucions més simples.

L'algorisme que utilitzarem es limita a comprovar per a tot el conjunt de caixes que conformen l'escena si tots els vèrtexs d'aquestes es troben en el costat extern d'algun dels plans que defineixen el frustum i en aquest cas descarta la caixa. Per determinar si un punt es troba en la part exterior de un pla podem utilitzar el signe de la distància entre aquests dos. Donat un pla $ax + by + cz + d = 0$ i un punt p arbitraris, és ben conegut que la distancia amb signe D entre ells ve definida per:

$$D = \frac{ap_x + bp_y + cp_z + d}{\sqrt{a^2 + b^2 + c^2}} \tag{3.16}$$

Es pot veure que el resultat serà positiu si el punt es troba en el semiespai al que apunta la normal, 0 si forma part del pla i negatiu altrament. A més, com que només ens interessa el signe, podem descartar la divisió per l'arrel quadrada obtenint la següent expressió:

$$D' = ap_x + bp_y + cp_z + d \tag{3.17}$$

Podem observar que la última expressió es pot computar com el producte escalar entre el vector que descriu el pla $f = (a, b, c, d)$ i el punt p en coordenades homogènies.

$$D' = f \bullet (p_x, p_y, p_x, 1) \tag{3.18}$$

La importància de l'observació anterior rau en el fet que el producte escalar és una operació estàndard que trobarem en qualsevol llibreria matemàtica de caràcter general que sovint es implementada utilitzant instruccions vectorials especialitzades. Aquestes instruccions permeten computar aquest producte escalar en un sol pas i per tant poden suposar un millor rendiment.

Una possible implementació del test de solapament enter el frustum i una caixa podria ser la següent:

Algorisme 6 Test de solapament entre una caixa i un frustum

```
function BoxInFrustum(B, F)
    for all f ∈ F do
        t ← 0
        for all p ∈ B do
            if f • (p_x, p_y, p_z, 1) > 0 then
                t ← t + 1
            end if
        end for
        if t = 0 then
            return false
        end if
    end for
    return true
end function
```

On B és el conjunt de punts que conformen la caixa a provar i F és el conjunt de plans del frustum.

Un punt important que no hem tractat encara és l'obtenció dels conjunts F i B del mètode anterior. Com hem vist abans, les caixes que utilitzem estan alineades amb els eixos. Aquest tipus de caixes es defineix per els seus punts màxim i mínim. Siguin b_{min} i b_{max} aquests dos punts, podem descriure B com el conjunt de punts generat per la combinatòria a nivell de les

components entre aquests dos. Així B queda definit de la següent forma:

$$\begin{aligned} B = \{&(b_{min_x}, b_{min_y}, b_{min_z}),\\ &(b_{min_x}, b_{min_y}, b_{max_z}),\\ &(b_{min_x}, b_{max_y}, b_{min_z}),\\ &(b_{min_x}, b_{max_y}, b_{max_z}),\\ &(b_{max_x}, b_{min_y}, b_{min_z}),\\ &(b_{max_x}, b_{min_y}, b_{max_z}),\\ &(b_{max_x}, b_{max_y}, b_{min_z}),\\ &(b_{max_x}, b_{max_y}, b_{max_z})\} \end{aligned} \tag{3.19}$$

Per a obtenir el conjunt de plans que conformen el frustum utilitzarem l'algorisme presentat a l'article [Gribb01]. Aquest algorisme extreu els plans directament de la matriu de projecció del la càmera. La definició exacta d'aquesta matriu és un conveni que fixa la implementació del marc de rasterització que s'utilitza i per tant l'algorisme és lleugerament diferent segons aquest marc. Nosaltres ens centrarem en la versió que utilitza matrius de projecció segons defineix OpenGL. Per agilitzar l'explicació ens limitarem a presentar els resultats que es presenten a l'article i no entrarem en el detall de com s'obtenen. Només dir que s'utilitza el conjunt de punts que queden a dins del frustum després d'aplicar-los-hi la matriu de projecció i que està codificat en aquesta matriu. A continuació enumerarem els paràmetres (a, b, c, d) que defineixen cada un dels plans $left$, $right$, $bottom$, top, $near$ i far que formen el frustum:

$$\begin{aligned} left &\begin{cases} a = m_{41} + m_{11} \\ b = m_{42} + m_{12} \\ c = m_{43} + m_{13} \\ d = m_{44} + m_{14} \end{cases} , right \begin{cases} a = m_{41} - m_{11} \\ b = m_{42} - m_{12} \\ c = m_{43} - m_{13} \\ d = m_{44} - m_{14} \end{cases} \\ bottom &\begin{cases} a = m_{41} + m_{21} \\ b = m_{42} + m_{22} \\ c = m_{43} + m_{23} \\ d = m_{44} + m_{24} \end{cases} , top \begin{cases} a = m_{41} - m_{21} \\ b = m_{42} - m_{22} \\ c = m_{43} - m_{23} \\ d = m_{44} - m_{24} \end{cases} \\ near &\begin{cases} a = m_{41} + m_{31} \\ b = m_{42} + m_{32} \\ c = m_{43} + m_{33} \\ d = m_{44} + m_{34} \end{cases} , far \begin{cases} a = m_{41} - m_{31} \\ b = m_{42} - m_{32} \\ c = m_{43} - m_{33} \\ d = m_{44} - m_{34} \end{cases} \end{aligned} \tag{3.20}$$

On m_{ij} indica l'element que es troba a la fila i i la columna j de la matriu de projecció de la que parlàvem. Els plans que s'obtenen amb aquest mètode utilitzant una matriu de projecció venen donats en espai de visió. Per obtenir el plans en espai de món, la qual cosa és imprescindible ja que les caixes estan definides en aquest sistema de coordenades, simplement, com s'explica a l'article, cal utilitzar una matriu que combini la matriu de projecció i la de visió. Essent aquesta matriu de visió la inversa de la transformació que experimenta la càmera en espai de món.

Transicions entre nivells de detall

Com hem vist anteriorment cada un dels nivells de detall està representat sobre un anell de geometria. Aquest anell inclou en el seu interior tota la geometria que representen els nivells inferiors a ell. En concret la de nivell inferior. Degut a la diferència en la densitat de mostres entre un nivell i l'altre, la frontera entre aquestes presentaran forats. Una forma de solucionar-ho consisteix en transformar les mostres del de més detall mesclant-les progressivament amb les del nivell amb menys detall. El nivell de progressió el podem computar com a $\alpha = max(\alpha_x, \alpha_y)$ on α_x es càlcul de la següent forma:

$$\alpha_x = clamp(|x - v_x| - (\frac{n-1}{2} - w - 1)/w, 0, 1) \tag{3.21}$$

On x és la posició del vèrtex, v_x és el desplaçament continuo de la càmera, n és el nombre de mostres de l'anell a la frontera exterior, w controla la distancia a la que comença la regió de canvi i $clamp(a, b, c)$ força a en el rang $[a, b]$. α_y és calcula anàlogament. A la figura **??** podem veure aquestes regions de transició.

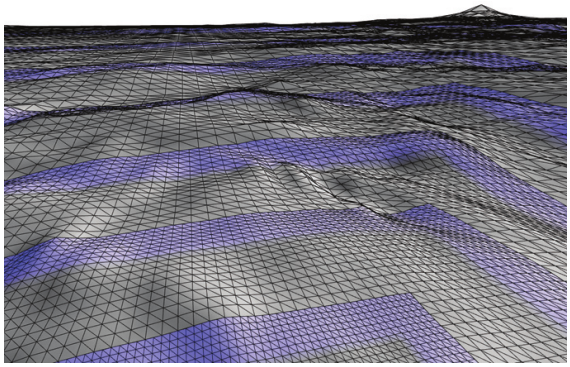


Figura 3.9: Zones de transició (en morat) entre dos nivells de detall. Imatge extreta de [Asirvatham05].

Per calcular el resultat final interpolarem linealment amb α l'alçada a el punt en qüestió amb l'alçada que hi ha en la següent nivell. A l'hora de determinar l'alçada del següent nivell al punt, podem detectar 2 casos. Existeix una mostra en el punt i en aquest cas no cal fer res o que aquesta mostra no existeixi (el següent nivell té la meitat de mostres). En aquest últim cas caldrà calcular el punt mig entre dos mostres simulant la triangulació de la malla de l'anella amb menys resolució. Aquest esquema també l'aplicarem a les normals per tal de suavitzar les transicions d'aquestes.

Capítol 4

Texturació del terreny

Per tal de donar una mica més de realisme i complexitat a l'escena és imprescindible poder aplicar algun tipus de textura a la geometria generada. Típicament, per cobrir grans extensions de superfície s'utilitza una textura que es va repetint cada cert espai. Aquesta textura té la peculiaritat de que els marges coincideixen, és a dir, que es poden posar dos instàncies de la textura una al costat de l'altra de forma que no es nota la frontera entre les dos. Aquesta solució, tot hi que és molt fàcil de implementar i és extremadament eficient, acostuma a produir patrons molt regulars que fan baixar la qualitat visual i el realisme de l'escena. Per intentar solucionar aquest problema es proposa la utilització de *Wang tiles*. Aquest mecanisme ens permet omplir el pla amb un conjunt de rajoles quadrades que poden encaixar entre elles de forma no-periòdica. Els marges de les rajoles, típicament etiquetats amb un color tal que dos costats amb el mateix color poden ser posats un al costat de l'altre sense que es noti la frontera, ens introdueixen certes restriccions en com es poden col·locar aquestes rajoles. D'aquesta manera dos costats no es poden tocar si no tenen marges amb colors coincidents. També cal dir que es prohibeix la rotació d'aquestes rajoles. En la figura 4.1 podem veure un exemple d'aquesta disposició per colors.

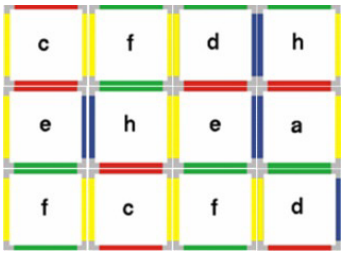


Figura 4.1: Exemple de la disposició de les rajoles segons els colors dels seus costats. Imatge extreta de [Cohen03].

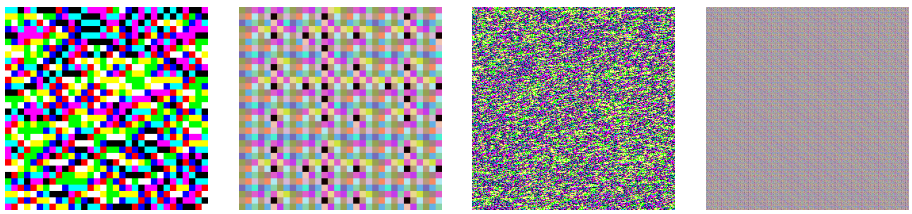
En les següents seccions parlarem de diferents estratègies que es poden seguir per omplir el pla de rajoles, com podem implementar-les en el nostre sistema i com sintetitzar, a partir de una imatge font, un conjunt vàlid de rajoles que s'adapti a l'estratègia escollida.

4.0.1 Aplicació de les Wang Tiles

A l'hora de aplicar les *Wang tiles* a la geometria que hem generat, és important veure quines limitacions ens podem trobar i quins requeriments hem de complir.

- la geometria es pot generar indefinidament. Això implica que la textura també ho haurà de fer.
- Les distribucions de rajoles han de ser repetibles, és a dir, donada una posició en el terreny hem de ser capaços de generar la mateixa rajola cada cop que sigui necessari.
- La textura ha de poder cobrir en tot moment l'espai que sigui visionat. En aquest punt cal destacar que els geoclipmaps que hem implementat tenen la possibilitat de cobrir extensions molt grans de terreny sense utilitzar gaires recursos. Com veurem més endavant, això ens limitarà.

Un dels articles referents en la generació eficient de distribucions *de Wang tiles* és [Cohen03]. Aquí és parla de com generar un conjunt de rajoles de forma que es pugui omplir el pla infinit de forma no periòdica amb un nombre reduït de rajoles. Aquest mètode s'allunya dels resultats que presenta la literatura teòrica en els que es busca el conjunt de rajoles més petit possible de forma que, a través de un procés determinista, s'ompli el pla de forma estrictament aperiòdica. El mètode presentat, en canvi, es recolza en la col·locació de rajoles de forma estocàstica seguint les regles que garanteixen la validesa de l'enrajolat. L'algorisme comença escollint una rajola del conjunt de forma aleatòria, seguidament és limita a col·locar rajoles, una a una, escollides uniformement del subconjunt de rajoles vàlides per la configuració creada fins el moment, d'esquerra a dreta per files. Cal remarcar que el conjunt de rajoles que s'utilitza té la propietat que el subconjunt de rajoles valides per a qualsevol configuració possible creada amb l'algorisme té una cardinalitat superior a 1. Aquest fet, de que sempre hi hagi la possibilitat d'escollir en cada pas almenys dos rajoles, garanteix la no periodicitat de la configuració final. Cal mencionar que aquest mètode no garanteix una aperiodicitat estricta cosa que si que garantirien els resultats teòrics. Això no resulta un problema ja que els resultats obtinguts amb aquesta tècnica són perceptiblement més aleatoris. En la figura podem veure una comparació entre una disposició feta amb aquest mètode i una disposició estrictament aperiòdica.



(a) (b) (c) (d)

Figura 4.2: Comparació entre distribucions de rajoles estocàstiques i estrictament aperiòdiques. (a) 32x32 estocàstic, (b) 32x32 aperiòdic, (c) 256x256 estocàstic, (d) 256x256 aperiòdic. Imatges extretes de [Cohen03].

Per diferents raons que mencionarem a continuació, aquest esquema no ens servirà. Com es pot veure, amb l'algorisme descrit anteriorment, per calcular una rajola qualsevol és necessari calcular tota la configuració fins a aquella. Això, en el nostre cas on hem de cobrir una extensió no limitada és impossible, cal conèixer la mida de la superfície a cobrir. Una solució que podríem pensar és dividir la superfície en trossos de mida coneguda. Això té diversos problemes. En primer lloc les cantonades entre fronteres dels trossos adjacents han de coincidir, cosa que, a priori, no és garantit amb la tècnica en qüestió. A més, si trobéssim un forma de fer coincidir aquestes fronteres, la

quantitat de dades que hauríem de mantenir per a poder texturar tota la superfície visible podria ser massa gran, sobretot si la mida de la rajola és petita.

Un cop vistes les limitacions que ens imposen els algorismes seqüencials com el descrit anteriorment, ens inclinem per implementar alguna tècnica que ens ofereixi accés aleatori en el sentit que el càlcul de qualsevol rajola sigui independent de la disposició de les altres al mateix temps que es respecten les restriccions de veïnatge. A [Wei04] trobem un mètode que fa exactament això. L'article presenta una forma de aplicar una textura virtual de dimensions arbitraries creada amb *Wang tiles*. Una de les peculiaritats de la tècnica és que no requereix guardar de cap forma la disposició de rajoles sinó que, a partir de la posisió en l'espai de món, calcula directament les coordenades en un atles de textures. Això és essencial en el nostre cas ja que, com hem vist abans, si les rajoles són petites, donades les dimensions de la zona que potencialment podem visualitzar, si haguéssim de guardar la disposició de rajoles, la quantitat de dades que hauríem de mantenir podria ser massa gran. Una altre avantatge de la tècnica és que és transparent (amb certes limitacions) als mètodes de filtrat de textura que proporciona el hardware gràfic. A continuació mostrarem amb més detall en que consisteix la tècnica, com implementar-la i com la podem adaptar a les nostres necessitats.

En primer lloc cal veure que si volem accés aleatori el nostre conjunt de rajoles ha de contenir totes les possibles combinacions. Així la cardinalitat d'aquest sera de ${K_v}^2 \times {K_h}^2$ rajoles, on K_v i K_h són el nombre de colors verticals i horitzontals respectivament dels marges de les rajoles del nostre conjunt. Típicament K_v i K_h són petits i per tant la mida del conjunt no hauria de resultar un problema massa notori. A més, el fet de tenir el conjunt complet ens proporciona maneres de col·locar les rajoles en un atles de textures de manera que les tècniques de filtratge de textura que ens proporciona el hardware funcionin de forma transparent. Més endavant aprofundirem en aquest tema.

En essència el que volem fer és donada una posició en el pla, trobar les coordenades referents a una rajola continguda en una atles de textures de manera que el resultat final compleix les restriccions de veïnatge imposades per els colors. Per fer això primer necessitem saber tant les coordenades de la rajola el espai de món $T = (T_h, T_v)$ com les coordenades internes d'aquesta $t = (t_h, t_v)$. Per això farà falta conèixer la mida de la rajola en espai de món. Aquesta és una dada que ve donada. Sigui S la mida de costat de la rajola quadrada i $P = (P_h, P_v)$ la posició en el pla del que volem computar, calcularem T i t de la següent forma:

$$\begin{aligned} T &= floor(\frac{P}{S}) \\ t &= fract(\frac{P}{S}) \end{aligned} \tag{4.1}$$

On $floor()$ denota la part entera (arrodoneix cap a $-\infty$) i $fract()$ denota la part fraccionaria.

Un cop coneguda T, voldrem conèixer els colors assignats a aquesta rajola. Per a fer-ho utilitzarem una funció de hash computacional implementada de una forma similar a la descrita a la

secció 2.0.2. D'aquesta forma calcularem C_S, C_E, C_N i C_W a partir de T de la següent manera:

$$\begin{aligned} C_S &= hash(hash(T_h) + T_v) \pmod K_v \\ C_E &= hash((T_h + 1) + hash(2 \times T_v)) \pmod K_h \\ C_N &= hash(hash(T_h) + (T_v + 1)) \pmod K_v \\ C_W &= hash(T_h + hash(2 \times T_v)) \pmod K_h \end{aligned} \tag{4.2}$$

Es pot veure que C_E i C_N es calculen amb la funció de hash de la rajola veïna a la calculada. Així es garanteixen les restriccions de veïnatge. També cal apreciar que el càlcul està basat en una funció caòtica anomenada *Cat-map* que es pot definir de la següent forma:

$$\begin{aligned} x' &= x + y \\ y' &= x + 2 * y \end{aligned} \tag{4.3}$$

Aquesta funció intenta minimitzar la simetria en diagonal i n'augmenta la aleatorietat.

Un cop coneixem els identificadors dels marges de la nostra rajola introduirem un sistema d'empaquetat per tal de poder indexar de forma correcta a l'hora d'utilitzar les rajoles. En el sistema que es presenta a [Wei04] es forcen les restriccions de veïnatge entre rajoles dins de l'empaquetat. D'aquesta forma es garanteix certa compatibilitat amb els sistemes de filtratge basats en nivells de detall que proporciona el hardware. Aquest sistema de filtratge es basa en mantenir múltiples copies de la imatge original cada una amb la meitat de resolució que l'anterior. Per reduir la resolució a la meitat el que es fa és reduir cada bloc de 2×2 pixels a un pixel el color del qual és la mitjana dels que formen el bloc. D'aquesta manera si garantim les restriccions de veïnatge entre rajoles en l'empaquetat, les fronteres entre rajoles quedaran ben representades en les successives reduccions.

Sense entrar en els detall de com s'obtenen els resultats, podem veure que la següent funció garanteix el veïnatge per una fila de rajoles.

$$TileIndex1D(e_1, e_2) = \begin{cases} 0, & e_1 = e_2 = 0 \\ {e_1}^2 + 2e_2 - 1, & e_1 > e_2 > 0 \\ 2e_1 + {e_2}^2, & e_2 > e_1 \geq 0 \\ (e1 + 1)^2 - 2, & e_1 = e_2 > 0 \\ (e1 + 1)^2 - 1, & e_1 > e_2 = 0 \end{cases} \tag{4.4}$$

On e_1 i e_2 són respectivament C_W i C_E de la rajola o C_N i C_S. Cal notar que de moment estem tractant l'empaquetat en una dimensió i per tant e_1 i e_2 seran els identificadors numèrics corresponents a la dimensió tractada (vertical o horitzontal). Cal remarcar que la validesa de la funció rau en el fet de que disposem de totes les combinacions possibles de rajoles. Per estendre l'empaquetat a les 2 dimensions només caldrà combinar el resultat vertical amb l'horitzontal de la següent forma:

$$\begin{aligned} (I_x, I_y) = TileIndex2D(C_S, C_E, C_N, C_W) = (&TileIndex1D(C_W, C_E), \\ &TileIndex1D(C_N, C_S)) \end{aligned} \tag{4.5}$$

Podem veure un exemple d'aquest empaquetat a la figura 4.3.

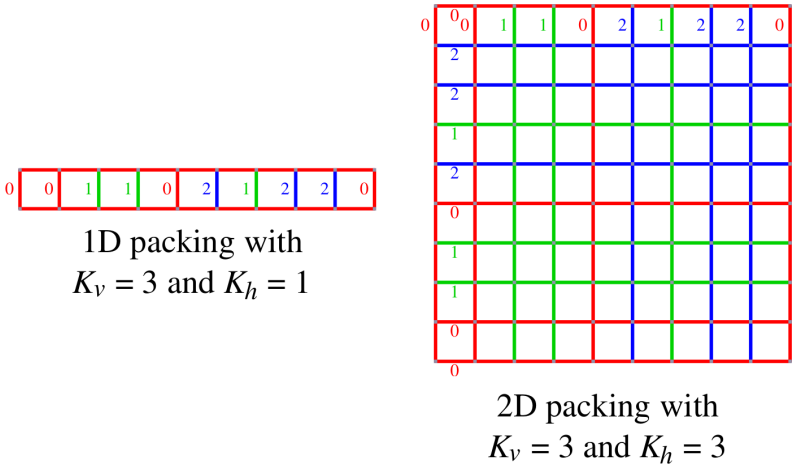


Figura 4.3: Exemple de empaquetat de rajoles utilitzant les equacions 4.4 a l'esquerra i 4.5 a la dreta. Imatge extreta de [Wei04]

Un cop tenim les coordenades I de la rajola a dins de l'atlas, per accedir al seu contingut utilitzarem t. Les funcions d'accés a textura esperen coordenades de l'atles normalitzades uv. Intuïtivament les podem calcular de la següent forma:

$$uv = \frac{I + t}{K^2} \tag{4.6}$$

On $K = (K_v, K_h)$. i K^2 el nombre de rajoles de constat de l'empaquetat.

Tot hi que hem dit que l'empaquetat és compatible amb les tècniques de filtrat basades en nivells de detall, aquestes, per defecte, assumeixen que les coordenades de textura són continues, cosa que es pot veure clarament que no és complirà en el nostre cas. Per a calcular quins nivells de detall s'han d'utilitzar per a obtenir la mostra de la textura filtrada corresponent a un cert pixel, s'utilitzen les derivades de les coordenades de textura respecte les coordenades de la pantalla en aquest pixel. D'aquesta forma es pot estimar el nombre de mostres que participen en el còmput del color del pixel i així escollir els nivells de detall que representen millor la aportació d'aquestes mostres. Això implica que els pixels que continguin les fronteres entre rajoles (on hi ha discontinuïtat en les coordenades) aquestes derivades seran grans i per tant s'utilitzarà un nivell de detall incorrecte. La solució a aquest problema consisteix en proporcionar les derivades directament a les funcions encarregades de obtenir la mostra filtrada. Aquestes derivades es poden calcular de forma discreta utilitzant funcions especials que proporciona el hardware gràfic, en aquest cas utilitzant les coordenades uv' calculades de la següent forma:

$$uv' = \frac{T + t}{K^2} \tag{4.7}$$

Com sabem, en teoria, el terreny al que aplicarem aquestes textures pot ser explorat indefinidament. Això pot comportar que utilitzant els mètodes que hem presentat abans per trobar la coordenada de la rajola, es presentin problemes de precisió numèrica. Això és per què la posició del pla serà representada en aritmètica de punt flotant. Es ven sabut que aquesta representació numèrica sacrifica precisió per tal de acomodar nombres grans. Una observació que podem fer

és que les coordenades de rajola en el terreny poden ser representades per enters si la seva mida (S) és entera. La representació d'enters en complement a 2 té precisió infinita (o exacta) i, per tant, ens agradaria explotar aquesta propietat. A més, no creiem que aquesta imposició suposi una restricció notòria. Igual que abans, necessitem dos coordenades. T, la coordenada de la rajola que en aquest cas serà representada per un enter i t, les coordenades internes de la rajola representades com a punt flotant. Un cop trobades aquestes coordenades el procediment és el mateix que el que hem descrit anteriorment i per tant la part que modificarem serà per trobar aquestes sense pèrdua de precisió.

La solució que plantegem consisteix en obtenir T i t de la malla en forma de graella dels geometrical clipmaps presentats a la secció 3.1.1. Les dades que utilitzarem d'aquesta graella seran la posició entera traslladada representada per enters en complement a 2 G_l i la posició no traslladada representada en punt flotant g_l. Podem veure que aquestes dades seran el més precises possibles. Les posicions enteres tenen pressió infinita mentre que els càlculs per obtenir les no traslladades són fets en un marc local. Cal notar que la graella de la que parlem està seccionada en múltiples nivells de detall a diferents escales, d'aquí el subíndex l que indica el nivell. Recordem de la secció 3.1.1 que l'escala de cada graella serà 2^l. En aquest cas tan g_l com G_l no venen multiplicades per aquesta escala però la translació soferta per G_l esta dividida per 2^l. Un primer pas que podem fer és calcular T a partir de les dades que hem presentat. És fàcil de veure que això ho podem fer de la següent forma:

$$T' = floor(fract(g_l) * 2^l) + G_l * 2^l \tag{4.8}$$

Cal notar que el resultat de l'expressió $floor(fract(g_l) * 2^l)$ ha de ser transformat a enter abans de operar i que T' és enter. Podem veure que T' indexa una graella unitària que dividint per S passa a ser de la mida de la rajola i per tant tenim que:

$$T = \frac{T'}{S} \tag{4.9}$$

Per obtenir I a través de T utilitzarem exactament els mateixos mètodes que abans. Ara, per calcular t amb aquest esquema utilitzarem la següent expressió:

$$t = T' \pmod S + fract(g_l) * 2^l \tag{4.10}$$

Cal notar que ara t calcula les coordenades interior de la rajola en el rang $[0, S)$ (no està normalitzat). Finalment per calcular les coordenades finals uv procedim de la següent forma:

$$uv = \frac{I * S + t}{S * K} \tag{4.11}$$

També podríem haver normalitzat t i procedir com abans.

Un punt que queda per aclarir és el de l'obtenció de G_l i g_l. Aquestes variables provenen de les dades que defineixen la graella triangulada. Com hem vist a en la secció 3.1.3 les dades que defineixen aquesta graella tenen la propietat de que el *provoking vèrtex* de cada cel·la (formada per 2 triangles) serà igual. Per tant, podem calcular G_l utilitzant *flat shading* amb les coordenades enteres de la graella desplaçades. Pel que fa a g_l només caldrà la interpolació de la posició sense traslladar dels triangles que formes la graella.

Una critica que es pot fer a la tècnica utilitzada és el fet de que les coordenades de textura no s'ajusten a la forma del terreny sinó que l'aproximen com si fos pla. Això pot ser un problema si el desnivell canvia molt abruptament produint una deformació visible de la imatge. Aquest problema és ben conegut i existeixen tècniques com la presentada a [Geiss08], on s'aproxima la superfície amb diferents plans per tal de trobar una millor representació, que el podrien solucionar. Caldria estudiar, però, la compatibilitat d'aquestes solucions amb el nostre sistema de textura per rajoles. De moment aquesta millora la deixarem per a futures revisions.

4.0.2 Generació de un conjunt de rajoles

En aquest apartat mostrarem com és poden produir conjunts de *Wang tiles* a partir de mostres de una imatge font. La tècnica presentada aquí esta basada en la que es descriu a [Cohen03] però l'hem modificat lleugerament utilitzant tècniques de [Efros01] per tal d'aconseguir una major variabilitat en el contingut generat.

L'algorisme que seguirem és força senzill. En primer lloc escollirem K_v i K_h mostres de la imatge de partida on K_v i K_h són el nombre de costats verticals i horitzontals respectivament del conjunt de rajoles que volem crear. Cal puntualitzar que necessitem dos conjunts de mostres, les mostres que formaran els costats verticals i les que formaran els horitzontals. Un cop seleccionades les mostres de cada conjunt construirem totes les ${K_v}^2 \times {K_h}^2$ possibles combinacions amb 2 mostres del conjunt K_v i 2 del del conjunt K_h disposades tal com podem veure en la figura 4.4.

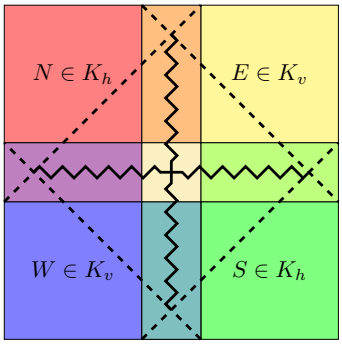


Figura 4.4: Formació de una rajola a partir de 4 mostres. 2 del conjunt K_v i 2 del conjunt K_h. Les linees en zig-zag representen talls generats amb l'algorisme de [Efros01] i els solapaments entre mostres es representen amb zones amb un color entremitg entre les dos. Les linees discontinues representen les linees per on cal retallar per obtenir la rajola final (sense rotar).

En la figura 4.4 és pot veure unes zones on les mostres es solapen. Aquestes zones les combinarem utilitzant l'algorisme de optimització presentat a [Efros01] que troba el tall amb error mínim entre dos mostres que es superposen. Aquest tall garanteix una certa continuïtat entre les dos mostres solapades. Per trobar el tall en una regió s'utilitza un esquema de programació dinàmica. Si B_1 i B_2 són dos blocs que es solapen verticalment amb les regions de solapament B_1^s i B_2^s respectivament, llavors la superfície d'error es defineix com a $e = (B_1^s - B_2^s)^2$. Si volem trobar el tall vertical amb error mínim d'aquesta superfície, la recorrerem des de $i = 2$ fins a la llargada d'aquesta i computarem l'error acumulat de tots els talls possibles de la següent manera:

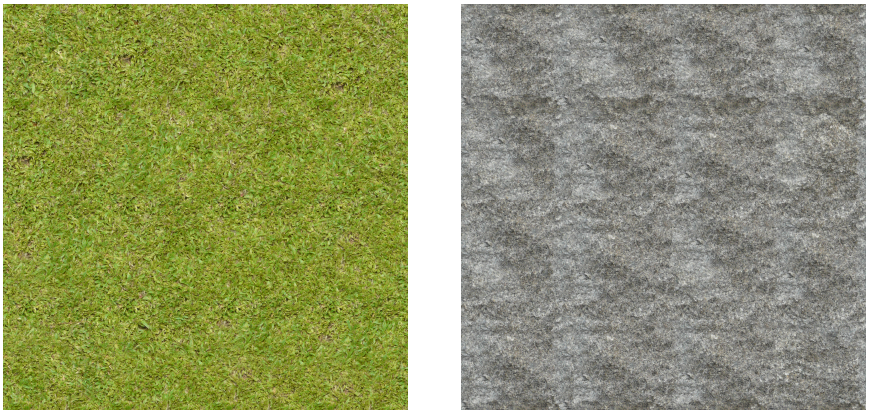
$$E_{ij} = e_{ij} + min(E_{i-1j-1}, E_{i-1j}, E_{i-1j+1}) \tag{4.12}$$

D'aquesta forma, el mínim obtingut a la última fila indica el mínim total i a partir del qual reconstruirem el tall seguint verticalment els mínims locals.

Un cop ajuntades les 4 mostres amb l'algorisme de [Efros01] retallarem per les seves diagonals de forma que ens quedi una figura en forma de diamant (quadrat rotat 45 graus) marcat amb línies discontinues a la figura 4.4. Com és pot veure fàcilment, si, per exemple, la mostra N de una rajola i la mostra S de una altra són la mateixa mostra, aquestes rajoles encaixaran verticalment, de la mateixa manera que amb les E i W en horitzontal. Finalment, per obtenir la rajola final, només farà falta rotar 45 graus en sentit horari i empaquetar-la en un atles de textura utilitzant les coordenades obtingudes amb l'equació 4.5.

Escollir les mostres pot no ser una tasca simple. A [Efros01] només cal escollir-ne una tal que minimitzi l'error del tall de solapament en el punt on toca col·locar-la. Així, un pot provar totes les mostres possibles i quedar-se amb la que doni millor resultat o una que tingui un error inferior a un llindar escollit. En el nostre cas les mostres són compartides per a un subconjunt de totes les rajoles, per tant, el que podria ser un bon candidat local en una rajola no ho sigui per a les altres. Cal pensar que la funció d'error a minimitzar és la suma de tots els talls de totes les rajoles del conjunt, si volguéssim trobar un mínim general la complexitat del problema incrementaria de forma molt notòria. És per això que s'ha decidit iterar sobre diferents conjunts escollits aleatòriament. El conjunt amb un error inferior és el que utilitzarem. Tot hi que aquesta forma de procedir pot semblar molt ingènua, els resultats obtinguts són prou bons per a l'ús que en volem fer. Una altra possibilitat seria fer una cerca més dirigida. Existeixen multitud d'esquemes per explorar espais de solucions de problemes complexos, per exemple *simulated annealing*, *hill climbing*, o algoritmes genètics. Aquests esquemes és probable que resultessin en una millor qualitat dels conjunts obtinguts amb el mateix nombre d'iteracions. Tot hi que, com hem dit abans, la opció escollida genera resultats suficientment bons, no es descarta utilitzar alguns d'aquest algorismes en futures versions.

Com es pot veure, amb aquest algorisme, tal com està descrit a [Cohen03], gran part de les rajoles comparteixen un tros considerable del contingut. Ho podem veure a la figura 4.4. Això, si els conjunts K_v i K_h són petits, pot suposar una repetició evident a l'hora de mostrar les rajoles. Per això es proposa una solució senzilla que consisteix en eliminar el contingut interior de la rajola i sintetitzar-ne un de nou amb l'algorisme de [Efros01] que ja hem vist abans. D'aquesta forma els marges de les rajoles continuen mantenint la seva coherència mentre que el contingut interior és més variat. Un exemple de atles de textura generat amb aquesta tècnica el podem veure a la figura 4.5



(a) (b)

Figura 4.5: 2 Atles amb un conjunt de wang tiles on $K_v = 4$ i $K_h = 4$

Cal dir que tot hi que el resultat de les rajoles sembla correcte de forma individual, quan es visualitza una combinació de rajoles en un pla, apareixen artefactes quadriculats. Creiem que això és degut a problemes inherents a l'algorisme de sintetitzat de [Efros01] i que podem veure reflectits a la figura 4.5b. Per a minimitzar-ne l'efecte es clau que l'elecció de les imatges continguin patrons isotròpics i homogenis o que l'escala d'aquest patrons (independentment de les seves propietats estadístiques) sigui petita comparat amb el de la mida de la rajola. Un exemple d'aquest tipus de textures podria ser el de la figura 4.5a.

Capítol 5

Diseny i ús de la libreria

En aquesta secció discutirem els aspectes més importants de la implementació en C++14 de la llibreria de renderitzat i generació de terrenys que hem creat. Aquesta part del document no només servirà per explicar el disseny i decisions preses en la implementació si no que també servirà a mode de referència d'us. Aquí no mostrarem els detalls interns de la implementació.

El disseny de la llibreria ha estat principalment motivat per pel fet de que volíem que fos el més general possible i ampliable. Al mateix temps ha de ser fàcil d'utilitzar, en el sentit de que no hauria de generar dubtes a l'usuari sobre la utilització correcta dels seus components.

La llibreria exposa tres components principals que volen ser abstraccions dels conceptes de renderitzat i creació de terrenys que hem mostrat en les seccions anteriors del projecte. Aquests components són:

- `TerrainDataCache`: Gestiona les dades que conformen el terreny local a la càmera,
- `TerrainModel`: S'encarrega de generar i organitzar les malles amb les que visualitzarem el terreny.
- `TerrainRenderer` Crea les imatges que representen el terreny descrit per els dos components anteriors.

Un petit diagrama en UML que descriu la relació entre aquest components podria ser el següent:

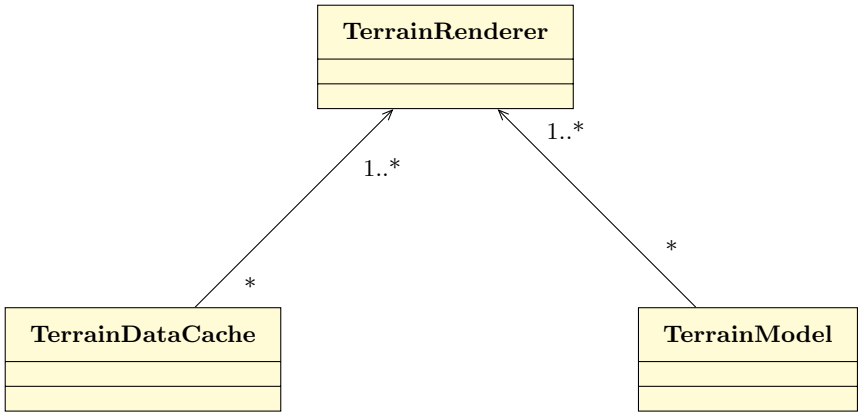


Figura 5.1: Diagrama UML de la relació entre els components principals de la llibreria

A continuació, per cada un d'aquest elements, farem una descripció del les motivacions que hi ha al darrera del seu disseny al mateix temps que en mostrem el seu ús correcte.

5.0.1 TerrainRenderer

El renderer és el component encarregat de generar imatges que representin el terreny. La seva funció principal consisteix en projectar en forma de imatge les dades del terreny que representen el model i la cache de dades. Com hem dit abans aquesta llibreria volem que sigui de caràcter general, per tant, cada un dels pixels de les imatges resultants no representarà el color final, si no una representació intermèdia del terreny que, per mitjà de un post processat, l'usuari pot transformar segons requereixi la seva aplicació. Aquest tipus de tècniques són més conegudes com a *deferred shading* i típicament s'utilitzen com a una optimització de processos d'il·luminació complexos ja que separen la complexitat de l'escena de la il·luminació. En aquest sentit, nosaltres aprofitarem aquesta idea de separar l'escena, en aquest cas el terreny, dels processos de shading, no tant com a optimització de cap tipus, sinó, com una forma de donar llibertat a l'usuari d'utilitzar el mètode de texturat i il·luminació que més li convingui. La sortida d'aquestes dades es farà per mitjà de buffers proporcionats per l'usuari en forma de textures i, per tant, aquest podrà escollir les dimensions i els formats que més l'interessin. A continuació enumerarem i descriurem les dades que exposarem a l'usuari. Noteu que aquestes dades composen els pixels de les imatges obtingudes després de transformar i projectar el terreny:

- `terrain_data`: Conté les dades del terreny. Estan formades per una vector de 4 elements que conté la normal de la superfície i l'alçada. Les components de la normal la trobem en els elements x , y i z del vector, mentre que l'alçada la trobem en la component w.
- `texture_coords`: Conté les coordenades normalitzades de textura a un atlas de wang tiles assignades. Estan formades per un vector de 2 elements que representen les coordenades u i v respectivament.
- `texture_coords_dF`: Conté les derivades parcials de les coordenades de textura respecte les coordenades de pantalla. Estan formades per un vector de 4 components on les components x i y representen les derivades respecte la component horitzontal ($dFdx$) i les components z i w representen les derivades respecte la component vertical ($dFdy$).
- `depth`: Conté els valors generats pel marc de rasterització en el test de profunditat en coordenades de finestra.
- `stencil`: Conté els valors generats pel marc de rasterització en el test d'stencil.
- `depth_stencil`: Conté una combinació dels valors `depth` i `stencil` descrits anteriorment.

Renderitzar totes aquestes dades de cop no sempre serà útil. Per exemple, si volem texturar el terreny de forma procedural, és evident que no ens voldrem malgastar recursos en generar coordenades de textura que no fan falta. És per això que definirem diferents tipus de renderers segons les sortides que tinguin actives. Cada implementació hauria d'utilitzar el mínim de recursos possibles per a proporcionar el conjunt de sortides demanades. Una solució naïf al problema seria implementar un renderer diferent per a cada subconjunt de sortides possibles. Això ràpidament podem veure que no és viable des del punt de vista de la mantenibilitat, inclús utilitzant les tècniques típiques de programació orientada a objecte. A més, podem veure que la computació de cada una de les sortides és molt ortogonal respecte de la resta. Això ens porta a pensar

en la utilització de mecanismes que permetin activar i desactivar parts del del programa segons convingui. És més, ens agradaria garantir certa correctesa, almenys pel que fa a les capacitats de cada tipus de renderer, abans de crear el programa, és a dir, en temps de compilació. Una de les possibilitats per aconseguir aquest tipus de comportaments és utilitzar tècniques de metaprogramació. Donat que el llenguatge de programació que s'utilitza és C++14 utilitzarem el sistema de metaprogramació basat en sistema tipus genèrics (o *template metaprograming*) que proporciona. En el passat, aquest tipus de tècniques s'han criticat apel·lant els llargs temps de compilació que comportaven i a la seva complexitat. Aquesta critica, tot hi que encara és en part vigent, cada cop està menys fonamentada. Per una banda, els compiladors han evolucionat de forma notable per solucionar els problemes de rendiment i per l'altra, l'estàndard cada cop incorpora més facilitats per tal de que aquestes tècniques semblin que formen més part del llenguatge i no pas una espècie de *hacks* amb el sistema de tipus només assequibles per experts en la matèria.

En primer lloc veurem com hem definit els tipus per tal de que pugui generar tota la combinatòria de sortides possibles. Per fer això simplement crearem una class template de C++ amb un paràmetre que definirà el subconjunt de sortides. Una class template es defineix com una família de tipus, és a dir, una entitat que accepta un nombre indeterminat de paràmetres i que descriu diferents tipus segons el valor d'aquests. Aquest paràmetres poden ser altres tipus, o, sense entrar en gaires detalls, alguns valors que es poden representar amb enters. Aquest paràmetre que hem dit que representa el subconjunt de sortides seria fàcilment representable com a una seqüència de bits de tal forma que si l'element i-èssim del conjunt està present, el valor del bit i-èssim és 1, altrament 0. Aquesta forma tan coneguda de representar subconjunts és fàcil de codificar amb un enter o, si volem una mica més de control sobre el rang i les operacions que es poden realitzar amb aquest subconjunt, un enumeració tipificada (`enum class` en C++). Aquest tipus d'entitat en el fons són enters, i per tant, les podem utilitzar per a descriure el subconjunt de sortides de la família de tipus que estem creant. La descripció exacta es pot trobar en el fitxer `RenderOutput.hpp` on es defineixen el `enum class RenderOutput`, i les típiques operacions de conjunts sobre aquest tipus: complement, unió, intersecció i la diferència simètrica. També s'hi descriuen utilitats per determinar la aritat del conjunt, si és subconjunt de un altre i es proporciona una llista ordenada (segons la seqüència de bits) amb tots els elements que poden formar un conjunt. Aquesta llista no només es útil per fixar un ordre entre els elements del conjunt, sinó que facilita alguns algoritmes de metaprogramació que els manipulen. Cal dir que totes aquestes operacions poden ser avaluades en temps de compilació, altrament, no seria possible utilitzar aquesta representació. En general, la forma més intuïtiva de descriure aquest conjunts de sortides és amb l'operació de unió. Un exemple real podria ser el següent:

```
constexpr RenderOutput o =
                  RenderOutput::texture_coords    |
                  RenderOutput::texture_coords_dF |
                  RenderOutput::terrain_data      |
                  RenderOutput::depth_stencil     |
                  RenderOutput::none;

OpenGLTerrainRenderer<o> renderer;
```

El renderer que hem implementat pràcticament no té estat modificable per l'usuari. Les úniques variables que es poden canviar són les que determinen els paràmetres relacionats amb les Wang Tiles i la transformació de la càmera. La definició d'aquestes funcions és la següent:

```
void set_mvp_matrix(const glm::mat4& mvp_matrix)
```

On `mvp_matrix` és la matriu que combina la transformació de visió i la projecció de la càmera. i,

```
template<RenderOutput O = Output>
std::enable_if_t<(O & RenderOutput::texture_coords) != RenderOutput::none,
void> set_wang_tiles(int32_t log2_Kh, int32_t log2_Kv,
                     int32_t log2_tile_size)
```

On `log2_Kh` i `log2_Kv` representen les mides del conjunt de rajoles i `log2_tile_size` s'autodefineix. Cal notar que aquesta funció només serà activa si `texture_coords` és present al conjunt de sortides.

La única altra funció és la principal del Renderer que òbviament és la funció de renderitzat. La declaració és la següent:

```
template<
    template<size_t> class F, class TerrainCache
>
void Render(const OpenGLTerrainModel& model,
            const TerrainDataCacheView<F, TerrainCache>& cache,
            const Framebuffer& framebuffer)
```

On, el `model` representa la malla que s'ha d'utilitzar per mostrar les dades proporcionades per la `cache` i `framebuffer` és el conjunt de textures on es presentaran els resultats, és a dir, la sortida. De la creació i configuració del `model` i la `cache` en parlarem en les seves seccions pertinents. Ara ens centrarem en el `framebuffer`.

El `Frambuffer` és una classe interna de `OpenGLTerrainRenderer` que assegura que els buffers de sortida que proporciona l'usuari siguin vàlids i els configura per tal de que puguin ser utilitzats per el Renderer. Aquest objecte només té dues interaccions amb l'usuari. La creació i el netejat del seus elements. La declaració de la creadora és a següent:

```
template<GLenum... TextureTypes>
    Framebuffer(const stgl::OpenGLTexture<TextureTypes, GL_TEXTURE_2D>&...
        textures)
```

Aquesta creadora es tracta de una *variadic function template* de C++14. Simplement dir que aquest tipus de construccions permeten declarar funcions amb un nombre indeterminat de paràmetres. En aquest cas, si el nombre de paràmetres no és el mateix que el nombre de sortides especificades al Renderer, el programa no compilarà queixant-se en forma de `static_assert` de C++14 indicant l'error. Per a cada textura es comprova que el format (`TextureType`) sigui adequat per la sortida a la que anirà destinada, en cas que hi hagi algun error, novament el programa no serà vàlid i el compilador mostrarà un error indicant el problema. Per poder raonar sobre la validesa dels formats de les textures donades és clar que haurem de definir algun orde ja que amb la definició no en tenim prou. Aquest ordre és el definit a `RenderOutput` i del que hem parlat abans. Tot i això per claredat el repetirem aquí:

```
using RenderOutputElems =
        RenderOutputList<RenderOutput::texture_coords,
                         RenderOutput::texture_coords_dF,
                         RenderOutput::terrain_data,
                         RenderOutput::depth,
```

```
                        RenderOutput::stencil,
                        RenderOutput::depth_stencil>;
```

Si un dels elements no forma part del subconjunt de sortides l'ordre es manté. Simplement queda eliminat de la llista. Els formats vàlids per a `texture_coords` , `texture_coords_dF` i `terrain_data`, són tots aquells formats de color de OpenGL 4.5 no comprimits que tinguin el mateix nombre de components que els vectors de dades de sortida descrits anteriorment: 2 per a `texture_coords` i 4 per a `texture_coords_dF` i `terrain_data`. Per les sortides `depth` , `stencil` i `depth_stencil`, Qualsevol dels formats especials respectivament suportats per a OpenGL 4.5 són vàlids. Cal afegir que les sortides `depth` i `stencil` no són compatibles amb la sortida `depth_stencil` per motius obvis. En cas d'error el programa no compilaria.

L'altra funció que presenta el Framebuffer és la de netejar les dades de les seves textures. Es descriuen 4 diferents definicions segons el tipus de sortida a netejar. Aquestes definicions són les següents:

```
template<RenderOutput O, class T>
std::enable_if_t<(   O != RenderOutput::depth
                  or O != RenderOutput::stencil
                  or O != RenderOutput::depth_stencil)
                  ,
void> Clear(std::array<T, 4>&& value)

template<RenderOutput O>
std::enable_if_t<O == RenderOutput::depth,
void> Clear(float value)

template<RenderOutput O>
std::enable_if_t<O == RenderOutput::stencil,
void> Clear(int value)

template<RenderOutput O>
std::enable_if_t<O == RenderOutput::depth_stencil,
void> Clear(float depth_value, int stencil_value)
```

Amb un exemple d'us veurem millor com funcionen. Imaginem que volem netejar el buffer de `depth` del Framebuffer `framebuffer` amb el valor 1:

```
framebuffer.Clear<RenderOutput::depth>(1.0f);
```

Com que hem especificat que volíem netejar el buffer de `depth` les altres definicions s'han desactivat. Per tant la única que queda és la que accepta els paràmetres específics del buffer de `depth`. En cas que el Framebuffer no tingués en el seu conjunt de textures la de `depth` el compilador emetria un error en forma de `static_assert`. Les altres variants funcionen igual, simplement modifiquen els arguments segons convingui per a les diferents sortides.

5.0.2 TerrainModel

El model és el component encarregat de generar, emmagatzemar i organitzar la malla triangulada que servirà per a poder visualitzar els camps d'altures generats. Tot hi que per a la realització del projecte hem utilitzat unes APIs gràfiques establertes (OpenGL), el disseny d'aquest component s'ha fet per tal de que sigui API agnòstic. Aquesta propietat, típicament és bona ja promou la reutilització del treball. A més, sovint, el codi generat amb aquest tipus de dissenys al cap és més

abstracte i permet raonar millor sobre el funcionament del sistema sense haver d'indagar en els detalls específics de cada API. Dit això passem a veure'n els detalls més importants.

La classe de C++ `TerrainModel` descrita al fitxer `TerrainModel.hpp` implementa la creació de la malla necessària per a visualitzar el terreny descrita a la secció 3.1.3. Com hem dit, volem ser API agnòstics i per tant no podem assumir gaire sobre el format del vèrtex i índex que la conformaran. Una solució comú en aquest tipus de casos és la de utilitzar codi genèric. En C++ això s'aconsegeix amb *class templates*. Tot hi que estem utilitzant el mateix mecanisme que en el renderer, la finalitat que en volem obtenir és força diferent. En aquest cas simplement volem desacoblar la implementació del format dels triangles. Un altre factor que ens lliga a la API és la gestió de la memòria gràfica on emmagatzemarem aquest triangles. Aquí una solució que creiem que és força elegant és la que es podria anomenar injecció de dependències estàtica. Consisteix en utilitzar el sistema de tipus genèric, del que tant hem parlat, i el sistema d'herència, abastament conegut en l'àmbit de la programació orientada a objectes, de forma conjunta per afegir funcionalitats en temps de compilació a una classe. Passem a veure'n la definició:

```
template<template<class, class> class DeviceMemoryAllocation,
         class IndexT, class Vec2T,
         class Vec2TA = GenericVec2Access<Vec2T>,
         class Vec2TO = GenericVec2Ops<Vec2T, Vec2TA>
        >
class TerrainModel : public DeviceMemoryAllocation<IndexT, Vec2T>
```

On `IndexT`, `Vec2T` són el format dels índex i els vèrtexos respectivament, `Vec2TA` i `Vec2TO` descriuen operacions d'accés als element del vector i operacions aritmètiques necessàries per a la generació de la graella. També es força que `IndexT` sigui enter per motius obvis. `DeviceMemoryAllocation` serà l'encarregat de reservar la memòria on es guardaran les dades que descriuen la malla. Podem veure que gràcies a l'operació d'herència entre objectes, `DeviceMemoryAllocation` i `TerrainModel` formen un agregat de forma que la funcionalitat del primer passa a ser també del segon. Per tant, l'implementador del renderer, que haurà de ser l'encarregat de popular els buits i que té coneixement del format del vèrtexs i índex i del model de memòria, podrà crear funcionalitats per a poder obtenir aquest recursos de memòria de la forma que més li convingui. L'únic punt on hi ha d'haver un consens clar és en la forma de enviar les dades generades per `TerrainModel` a el `DeviceMemoryAllocation`. En aquest cas la forma més general possible és a través de la creadora d'aquest últim.

Un cop concretats els elements genèrics pel programador del renderer, a l'usuari només li cal crear l'objecte especificant la mida de costat i la mida de la post transform cache per la qual vol optimitzar. Vegem la declaració de la funció creadora.

```
  TerrainModel(Pow2<uint32_t>, ring_size, uint32_t post_transform_cache_size = 0)
```

Noteu que l'argument de la mida de la cache és opcional, si no s'especifica res o el valor d'aquest és 0, no s'activaran les optimitzacions. `Pow2` simplement és un tipus que força que el valor de l'argument sigui potència de 2. Si no ho fos, es transformaria automàticament en la potència de 2 inferior més propera. En general, forçar propietats d'aquesta manera no només garanteix que es compleixen unes precondicions, sinó que documenta l'existència d'aquestes.

5.0.3 TerrainDataCache

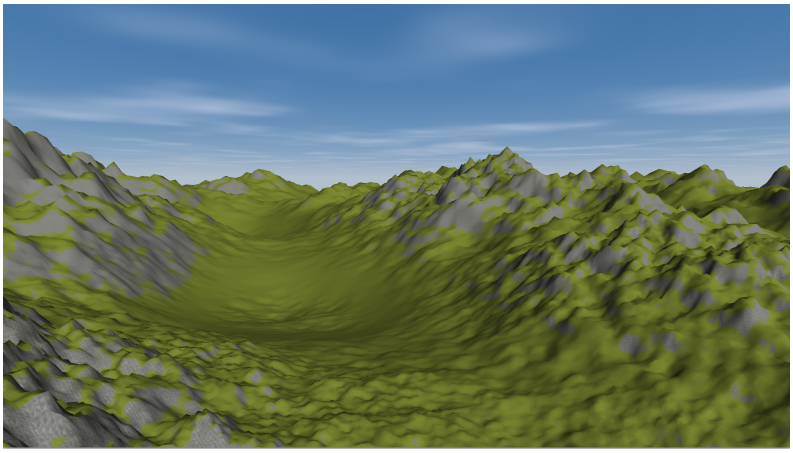
La cache és el component encarregat de proporcionar les dades que conformen el terreny local a la càmera. Des de un primer moment la motivació principal a l'hora de dissenyar aquesta part de la llibreria era que no estigués lligat a una implementació concreta. Per exemple, nosaltres, com hem vist abans, volem generar les dades a partir d'algorismes, però una altra possibilitat seria obtenir-les de una base de dades. Aquest detalls no són importants a l'hora de visualitzar el terreny i, per tant, volem que siguin transparents de cara a la interacció amb els altres elements de la llibreria. La solució més típica en aquest casos és la de una façana, és a dir, s'estipula un seguit de funcionalitats que les implementacions han de complir i l'usuari accedeix a les diferents implementacions ja sigui per mitjà d'abstraccions o, en el nostre cas, amb tècniques de codi genèric. Com hem vist el la secció 3 aquesta cache està composada per múltiples capes de detall. Sovint pot ser útil desactivar un subconjunt d'aquestes. Una solució que ens sembla elegant per a fer-ho és a través de vistes. Aquestes vises simplement són referències a un objecte que modifiquen la visibilitat d'aquest. La mida de la cache es pot conèixer en temps de compilació. Aprofitant aquest fet, hem decidit crear vistes estàtiques. Aquestes vistes accepten una expressió en forma de filtre que escolleix quines capes de la cache són visibles. Cal notar que tot això es determina en temps de compilació i per tant el filtre ha de poder ser avaluat en aquest estadi. Es evident que aquesta vista tindrà la mateixa façana que l'objecte real. La interfície que es presenta de cara a l'usuari és la següent:

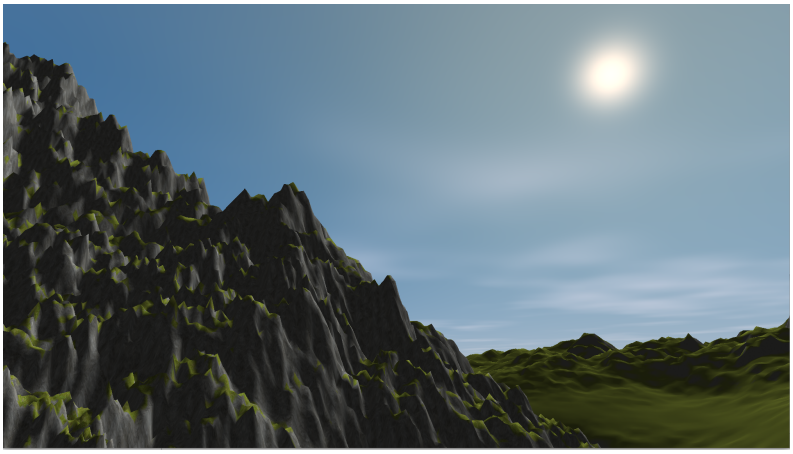
```
auto GetDataAt(const vec2_t& p)

void Lod(unsigned lod)

void Translate(float delta_x, float delta_y)

void Init(float x, float y, uint32_t lod = 0)
```

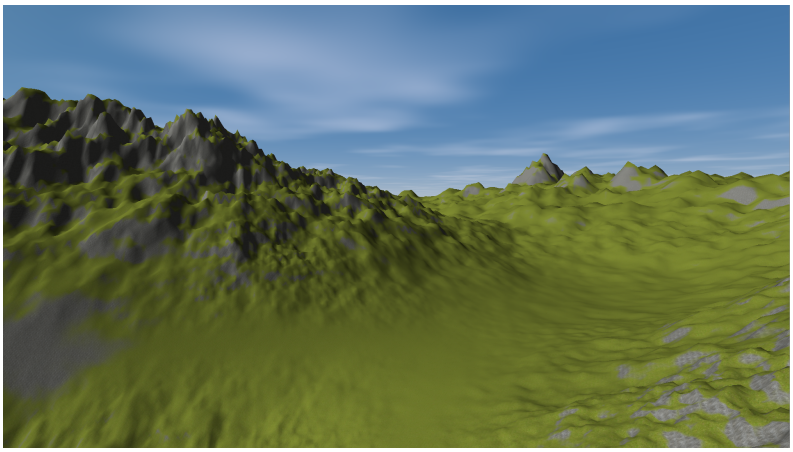
Com podem veure tenim les operacions bàsiques per navegar de forma incremental el terreny. `Lod()` modifica el nivell de detall base. `Translate()` i `Init()` s'autodefineixen. `GetDataAt()` retorna les dades en la posició indicada que poden ser utilitzades per a computar col·lisions o computar el nivell de detall necessari. Cal notar que utilitzant deducció automàtica del tipus de retorn, desacoblem el format de les dades de la façana.

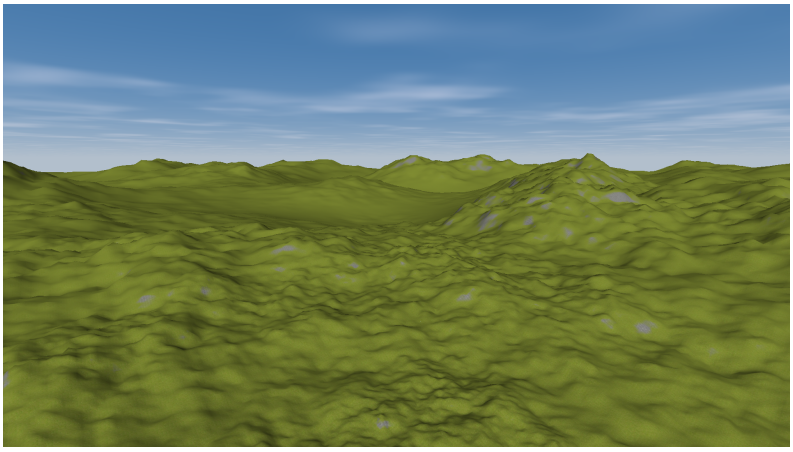
5.0.4 Aplicació de Mostra

Per provar i fer una mostra de les possibilitat de la llibreria hem creat una petita aplicació que permet explorar de forma indefinida un terreny generat. Aquesta aplicació permet controlar una càmera de forma lliure utilitzant el ratolí per escollir la direcció en la que apunta la càmera i les tecles W i S per desplaçar en la direcció a la que aquesta apunta. També es permet fer un moviment lateral amb les tecles A i D. A continuació mostrarem un seguit de imatges extretes de l'aplicació:









Cal dir que totes les imatges s'han capturat utilitzant els mateixos paràmetres en la funció de generació del terreny.

Aquesta aplicació ha estat programada en un entorn ArchLinux utilitzant els llenguatges C++14, CUDA i GLSL. Les dependències que té el programa són: OpenGL 4.5, CUDA 8, GLFW3, OpenImageIO i GLM. Es pot trobar el codi font a `https://bitbucket.org/gslm/gct`

A part de l'aplicació principal, afegim un programa que permet generar atles de rajoles de Wang. Aquest programa accepta un seguit de opcions que enumerarem a continuació on els elements marcats amb `[]` són opcionals:

- `-i`: Ruta a la imatge font.
- `-t`: Mida en pixels de les rajoles generades.
- `-b`: Mida en pixels dels marges de les rajoles.
- `-p`: Mida en pixels del solapament entre les mostres que formen part dels marges.
- `-v`: Nombre de colors verticals del atles generat.
- `-h`: Nombre de colors horitzontals del atles generat.
- `-d`: Nombre de mostres utilitzades per sintetitzar el contingut de les rajoles.
- `[-c]`: Nombre de iteracions permeses per a trobar el millor conjunt de rajoles.
- `[-f]`: Nombre de iteracions permeses per a trobar la millor mostra per sintetitzar el contingut de les rajoles.
- `[-s]`: Seed d'aleatorietat.
- `[-o]`: Ruta al fitxer de sortida.

Aquest últim programa ha estat escrit en C++14 i té com a dependències GLM i OpenImageIO. El codi font el podem trobar a `https://bitbucket.org/gslm/wtg`.

5.0.5 Resultats

Per avaluar el rendiment d'aplicacions gràfiques normalment s'utilitza el nombre de frames per segon que aquesta es capaç de computar. Aquesta mesura, tot hi ser un bon indicador de la fluïdesa que experimentarà l'usuari final, la seva obvia no linealitat respecte el temps és poc intuïtiva des del punt de vista computacional. És per això que per a la valoració final del rendiment utilitzarem el temps de generar una imatge (o frame). Per tenir una referència, els 30 frames que podríem dir que són el mínim acceptable en una aplicació interactiva impliquen a un temps de frame de 33 mil·lisegons. Si volem un llistó una mica més alt, 60 frames per segon són un temps de frame de 16 mil·lisegons.

En la nostra aplicació el temps de frame pot variar notablement de un frame a l'altre. Això dependrà principalment de la quantitat de geometria que sigui visible per la càmera i de la quantitat de mostres del terreny que s'hagin de generar. Per tant, si volem donar una idea clara del comportament de l'aplicació, és imprescindible fer alguna mena d'estudi estadístic. Sovint, aquest estudi es limita a donar un màxim, un mínim i una mitjana. Aquesta forma de procedir, tot hi semblar bona i que s'utilitza extensament, pot donar una imatge errònia del comportament de l'aplicació. Per exemple, imaginem una aplicació on el cost de computar un frame depèn de la paritat seqüencial d'aquest, els frames parells són molt barats de calcular mentre que els senars són extremadament costosos. En aquest escenari tindrem un mínim molt baix, un màxim molt alt i la mitjana ens donarà un punt entremig. Amb aquestes dades no podem saber si aquests valors mínim i màxim són outliers i hem de quedar-nos amb el valor central o realment tenim una disparitat molt gran entre el cost dels frames. Cal dir que l'exemple que hem posat simplifica alguns aspectes, però creiem que és prou il·lustratiu per descriure el problema al que ens enfrontem.

La solució proposada és la de utilitzar més dades en forma de histograma. Aquesta forma de procedir ens revelarà de forma visual la variabilitat del cost de frame. Si es donen molts casos on es superen els temps mínims establers, encara que la majoria de mostres es tinguin a dins dels límits haurem de concloure que l'aplicació no s'executa de forma fluïda. Una critica que podem fer a aquesta metodologia és el fet de que els frames menys costosos tenen més possibilitats de generar-se. Això seria un problema si la variabilitat fos molt gran. En el nostre cas aquesta variabilitat no és excessiva i per tant no actuarem per a solucionar-ho. En tot cas ho tindrem en compte en la anàlisis dels resultats si s'escau.

Per a computar aquest histograma utilitzarem el temps dels 10 primers segons. Aquest període temporal creiem que és suficient per tenir una idea prou clara del comportament que té l'aplicació. En el nostre sistema això suposa un total de uns 1000 frames. Per tal de que els resultats siguin mínimament repetibles descriurem una seqüència de moviments que cobreixin els diferents escenaris que es poden donar. Aquesta forma de procedir és molt criticable ja que no és del tot rigorosa: el resultat final dependrà de d'interpretació de l'usuari de la seqüència de moviments. La solució correcte seria implementar alguna forma de definir una animació de la càmera que sigui persistent entre execucions. A continuació enumerarem la seqüència de moviments que hem utilitzat:

1. Enfocar la càmera cap a baix i esperar 1 segon.

2. Ascendir, pressionant la tecla S, durant 3 segons de manera que la càmera apunti cap a

baix.

3. Rotar la càmera de forma que quedi mes o menys en posició horitzontal però lleugerament inclinada cap a baix i esperar 1 segon.

4. Avançar, pressionant la tecla W, durant 5 segons.

El terreny que que utilitzarem està format per 8 nivell de detall de 512×512 mostres. Aquest terreny, en la seva totalitat, està format per uns 5 milions de triangles. La resolució del les imatges que generarem serà de 1280×720 pixels. També cal especificar que la velocitat de moviment de la càmera és de 300 unitats per segon. Aquesta velocitat ha estat escollida a l'alça per tal de forçar l'actualització dels nivells de detall grans. Cal dir que els temps obtinguts només comprenen la part de renderització referent a la geometria i per tant no té en compte el temps de render del cel. En la figura 5.2 mostrarem l'histograma de les mostres obtingudes. També proporcionem un resum estadístic de les dades que el formen a la taula 5.1 i un gràfic que mostra l'evolució seqüencial del temps de frame a la figura 5.3.

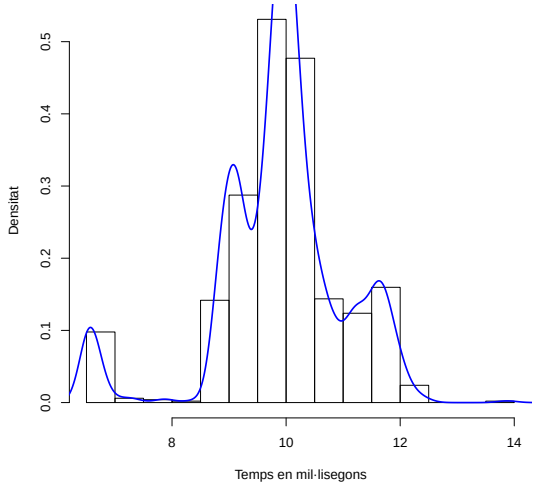


Figura 5.2: Histograma dels temps de frame format per 1000 mostres dividit en 15 intervals. En blau es mostra la corba de densitat.

min	1er Qu	mediana	mitjana	3er Qu	max
6.521	9.388	9.936	9.905	10.395	13.875

Taula 5.1: Resum estadístic de les dades que formen l'histograma de la figura 5.2

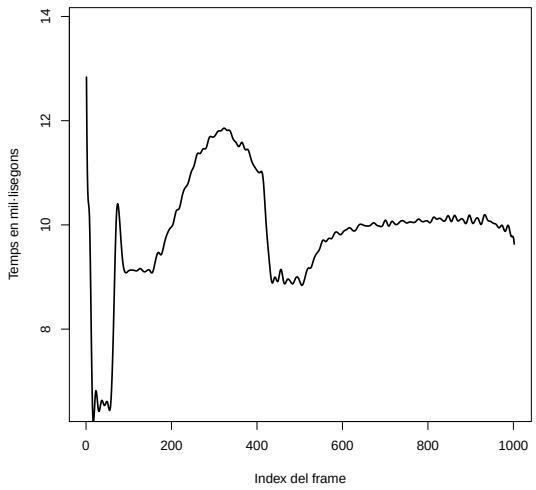


Figura 5.3: Evolució seqüencial del temps de frame de 1000 mostres.

Aquestes dades han estat obtingudes en un sistema Linux (Archlinux) amb un processador Intel i7-4702HQ CPU @ 2.20GHz, memòria ram de 16Gb DDR3 @ 1600 MHz i targeta gràfica Nvidia Quadro K1100M.

La primera observació que podem fer és que tots els resultats estan per sota dels llindars de interactivitat que hem descrit anteriorment així complit amb els requisits establers. Una altra observació és el fet de que el resultat no segueix estrictament una distribució normal. Això és d'esperar degut als diferents escenaris que es donen amb les instruccions que hem utilitzar per recollir les mostres. Tot hi això podem veure que una gran part de les mostres rauen a prop de la mitjana la qual cosa és un bon indicador de l'estabilitat en el cas general, es a dir, quan s'avança endavant. Noteu que és en aquest mode en el que estem més estona i per tant hi haurà més mostres. Aquest raonament ve recolzat per la figura **??** on podem veure que els últims frames tenen un temps de generació molt similar, al voltant de 10ms. A la figura **??** també podem veure que durant l'ascensió amb la càmera mirant a baix és quan més ha baixat més el rendiment, fins a 12 ms. Això creiem que es lògic ja que és en aquest estadi que el que hi haurà mes geometria visible. També cal observar que el rendiment és millor quan la càmera no es desplaça, amb un temps de 9ms.

Capítol 6

Conclusió

En aquest treball de final de grau s'ha presentat la implementació de una petita llibreria de caràcter general implementada en C++14 de generació i visualització de terrenys no acotats en temps real. La llibreria implementa amb el marc de renderitzat OpenGL 4.5 una variació de la tècnica de visualització de camps d'altures amb diferents nivells de detall presentada a [Asirvatham05]. La tècnica original obté les dades que representen el terreny de una base de dades. Això implica una quantitat no menyspreable de transaccions entre la CPU i la GPU. Nosaltres generarem de forma procedural amb funcions derivades del soroll fractal totes les dades directament a la GPU utilitzant el marc de còmput paral·lel CUDA 8. D'aquesta forma no només ens estalviarem totes aquestes transaccions, sinó que podrem accelerar els algorismes generadors aprofitant el seu alt nivell de paral·lelisme. A més, utilitzant idees de [Wei04] es proporcionen mecanismes per a generar coordenades de textura en atles de rajoles de Wang de forma que no es generin patrons repetitius . Aquesta part del projecte ve acompanyada amb una aplicació implementada en C++ que sintetitza aquest atles a partir de imatges font de forma similar a com es fa a [Cohen03]. En el nostre cas, però, s'intenta augmentar la variació del contingut de les rajoles amb l'algorisme de síntesi presentat a [Efros01]. Per tal de provar i presentar una mostra de les possibilitat de la llibreria, s'ha implementat una petita aplicació que permet navegar per un terreny no acotat texturat amb wang tiles.

El rendiment de la llibreria és relativament correcte, dins del que es pot considerar temps real. Els terrenys generats són visualment plausibles i presenten certs elements típics de la orografia real: heterogeneïtat i certs tipus de formes que ressemblen les esculpides per alguns processos d'erosió (pics escarpats i valls suaus). Una punt flac del projecte és en la generació i de les rajoles de Wang. Tot hi que eviten patrons repetitius com s'havia previst, introdueixen altres artefactes que poden fer minvar el realisme. Un altre punt a tenir en compte es el fet de que la llibreria presenta un model de renderitzat basat en *deferred shading* que permet a l'usuari escollir els models de il·luminació i pintat que més li continguin. A més, amb la llibreria hauria de ser possible implementar diferents algorismes com per exemple, la obtenció de un mapa de ombres.

Durant el desenvolupament del projecte hem agut de deixar algunes idees com a treball futur. A continuació farem una enumeració dels punts que creiem que són clau per a futures revisions del la llibreria:

- Cal estudiar si les optimitzacions de la malla per a hardware gràfic modern que hem insinuat

en el capítol 3 funcionen i en cas afirmatiu modificar o ampliar l'algorisme de generació de la malla per a tenir-les en compte.

- Cal estudiar alguna forma de solucionar la deformació que pateixen les coordenades de textura en condicions verticals per tal de millorar-ne l'aparença visual tal com s'indica al capítol 4
- Seria interessant que l'usuari pogués escollir diferents tipus de funcions generadores de terreny.
- Seria interessant intentar integrar la llibreria en algun motor gràfic per tal de facilitar-ne l'adopció.

Capítol 7

Informe de sostenibilitat

Aquest capítol està dedicat a l'anàlisi de sostenibilitat del treball final de grau (TFG) presentant en els capítols anteriors. L'organització d'aquest informe està basada en el document adjunt en la rúbrica d'avaluació del projecte de la qual la realització d'aquest anàlisis n'és un apartat. Seguint aquesta organització dividirem el capítol en tres seccions: L'estudi de l'impacte econòmic, l'ambiental i el social. Tal com s'indica al mencionat document, el contingut proposat és de caràcter general i, per tant, no sempre serà viable en el context d'aquest TFG. És per això que alguns dels temes indicats s'han omes o no s'han profunditzat tant com se'n dedueix del document. En tot cas, seguint les indicacions del document, aquestes simplificacions sempre intentaran ser degudament justificades. Abans de fer l'estudi pròpiament dit, és necessari fer una petita introducció del context en el que s'ha desenvolupat el projecte. Aquest context és clau a l'hora d'entendre algunes de les desviacions que em adoptat respecte les indicacions donades.

En primer lloc, cal dir que el resultat del projecte és una llibreria que volem que sigui distribuïda com a codi obert no restrictiu. En general, aquest tipus de llicències permeten a qualsevol interessat modificar i/o utilitzar el codi publicat sense restriccions sempre hi quan es mencioni l'autoria del treball inicial. En segon lloc, també cal dir que el treball ha estat realitzat durant un període de temps relativament llarg de forma força discontinua i per tant qualsevol previsió i planificació que s'hagués pogut fer en els apartats de GEP corresponents queda força allunyada de la realitat. És més, durant el període de temps entre que es va realitzar el GEP i l'escriptura d'aquest document sembla que el contingut de GEP ha estat modificat lleugerament i per tant alguns dels punts del document referents al GEP han estat omesos.

7.1 Estudi de l'impacte econòmic

Com hem dit, el projecte consisteix en una llibreria que serà distribuïda de forma pública amb una llicència de codi obert no restrictiva. Aquesta forma de distribuir el software és fàcil de veure que no genera ingressos directes. No obstant, es poden donar casos on l'usuari no tingui les competències necessàries o el temps i volgués integrar o ampliar aquesta peça de software per el seu benefici. Aquest casos podrien comportar ingressos en forma de consultories. Cal dir però que donada la simplicitat de la llibreria que hem creat és poc probable que aquests casos es donin.

En el context hem explicat que aquest projecte ha estat realitzat de forma intermitent en el

transcurs de un període de temps relativament llarg. Podríem dir que ha estat realitzat en el temps lliure de altres activitats. Per tant, és difícil descriure despeses que són exclusives de la realització d'aquest TFG. És evident que per a realitzar aquest projecte es necessari un lloc de treball amb els seus equipaments (connexió a interent, electricitat, etc.) i un maquinari que compleixi els requisits mínims establerts, però aquest elements, que suposarien una despesa, no han estat adquirits per fer el TFG i per tant no creiem que s'hagin d'incloure e la llista de despeses. Al mateix temps, com hem dit, el temps emprat no es pot considerar productiu, és a dir, no s'ha utilitzant temps destinat a altres activitats que podrien generar un benefici, per tant, tampoc creim lògic posar un cost econòmic a aquest.

Podem concloure que aquest projecte, tot hi que no podem veure que pugui generar cap ingrés, el cost de produir-lo ha estat virtualment nul i per tant podria ser considerat un projecte econòmicament viable. A més, el fet de poder considerar el cost del projecte zero, ens lliura de haver de pensar en qualsevol risc econòmic o imprevist.

7.2 Estudi de l'impacte social

Com hem dit, el resultat del projecte hauria de ser distribuït com un llibreria de codi lliure. Això implica que el codi està disponible a tothom i, per tant, pot ser utilitzat no només per la seva funcionalitat, sinó també com a objecte d'estudi. Creiem que aquest és l'impacte social més gran que es pot despendre del nostre TFG. En aquest sentit també podria reduir el temps de desenvolupament de solucions similars, ja sigui adoptant idees presents en el projecte o construint a sobre d'aquestes.

Es difícil preveure el rumb que un projecte de codi obert pot seguir. El més probable és que no sigui ni utilitzat ni estudiat per molta gent i per tant el seu impacte sigui mínim. Un altre escenari seria que aquest tingués èxit i la gent utilitzés i contribuís en el projecte i per tant tingués un impacte més alt. Portant a l'extrem aquest últim escenari es podria donar que l'evolució d'aquesta solució esdevingués una solució prou bona com per competir amb solucions comercials. En aquest cas, tant podria incentivar la millora d'aquest productes comercials així generant un impacte social positiu com el contrari.

El projecte també ha tingut un impacte personal no menyspreable. En primer lloc cal dir que, en part, la realització del projecte s'ha utilitzat com a excusa per provar tècniques i idees, sobretot de programació i de desenvolupament de software, que possiblement no hagués pogut utilitzar en un marc diferent del de un projecte com aquest. Al mateix temps, aquesta cerca de noves tècniques també ha estat font de frustracions personals, ja sigui perquè aquestes estaven per damunt de les meves possibilitats, o simplement perquè no funcionaven com s'havia previst. Com hem dit, aquest projecte s'ha realitzat de forma discontinua durant un espai temporal relativament ampli. De fet, s'ha començat de nou fins a 3 vegades per tal de satisfer un marc mental de com havia de ser el projecte que possiblement estava fora de les meves possibilitats reals, sovint ignorant iterar sobre solucions menys complicades les qual podrien incentivar la motivació per seguir amb el projecte.

7.3 Estudi de l'impacte ambiental

En general, el software d'us domèstic no representa un impacte ambiental directe que sigui fàcil de mesurar. És possible que l'eficiència dels algoritmes (minimitzar transaccions de memòria, cicles de processador, etc.) pugui reduir el consum elèctric mitjà dels components de un ordenador. Dit això, calcular l'impacte que aquesta peça de software tindrà en aquest sentit depèn de multitud de factors força imprevisibles. En primer lloc el volum d'ús del software generat amb la llibreria que, com hem vist en la secció anterior, tot hi que es pronostica baix, al ser codi obert, no hi ha una forma de saber com aquesta s'acabarà utilitzant. Després, el consum entre diferents components (models i marques) pot ser extremadament variable. Una altra qüestió que ens podem plantejar és si l'ús de la nostra solució en substitueix altres que tinguin un consum diferent, ja sigui a l'alça o a la baixa. Sumat a tot això hem d'afegir que tampoc coneixem la font d'energia de la que s'alimenten aquests components. En tot cas és extremadament difícil sinó impossible fer un càlcul, ni que sigui aproximat, de l'impacte ambiental en aquest aspecte.

Una altre impacte ambiental, més indirecte, que podríem identificar es el creat per la renovació del hardware per part dels consumidors finals. És conegut que la fabricació de hardware nou i les deixalles electròniques produïdes per la substitució d'aquest poden suposar un problema ambiental. Una forma de frenar aquest impacte podria ser el de suportar hardware relativament antiquat per tal de no incentivar al consumidor a renovar el seu maquinari. En aquest sentit podem dir que les tecnologies utilitzades per a la realització d'aquest TFG són majoritàriament suportades per a hardware de fa almenys 4 anys i s'espera que sigui suportat per hardware futur.

Com hem dit abans, el resultat d'aquest projecte vol ser publicat en forma de codi obert. Per tal de presentar aquest codi al món, és imprescindible que aquest sigui accessible a través de la xarxa. Això implica haver de mantenir un servidor on allotjar aquestes dades la qual costa podria tenir una petita petjada ambiental en forma de consum elèctric. Sovint el que es fa, i el que hem fet, és utilitzar serveis que proporcionen una plataforma pensada especialment per a distribuir codi com pot ser Github (`github.com`) o Bitbucket(`bitbucket.org`) i per tant mesurar la petjada exacta de una contribució és una tasca força complicada. Aquesta petjada dependrà principalment de la política energètica de les empreses al darrere d'aquest serveis. Aquestes polítiques no només regulen la font de energia utilitzada, sinó que també tenen en compte el consum de l'equipament i l'eficiència amb que s'utilitza aquest. En tot cas, la interacció amb aquests serveis és força reduïda, a més, podem assumir que aquestes plataformes no dediquen recursos a usuaris concrets sinó que són compartits per a tots. Això ens fa pensar que la petjada de la que estem parlant és minúscula.

Capítol 8

GEP

8.1 Introducció

En aquest document mostrarem l'abast del projecte tiulat "Generació procedural de terrenys potencialment infinits utilitzant CUDA". Concretament descriurem quins són els objectius del projecte, quin serà el seu abast i els problemes més immediats que ens podríem trobar. També farem una descripció dels requisits que ha de complir.

8.2 Formulació del problema

En el món dels videojocs sovint es necessari visualitzar grans extensions de terreny, ja sigui per què es vol representar un món on el jugador té una gran llibertat de moviments o simplement per què es vol crear un món infinitament explorable.

Per tal de poder construir aquests models de una forma prou realista una de les millors opcions és utilitzar algorismes que simulen els processos naturals que formen la orografia real. Aquests algorismes es basen en la generació de mapes de altures a partir de certs tipus de soroll i el post-processat d'aquests mapes. Aquesta tasca és extremadament paral·lelitzable cosa que motiva a la utilització de alguna plataforma paral·la, en aquest cas CUDA (Compute Unified Device Architecture).

Com hem dit abans es busquen terrenys amb un cert realisme, per tant l'aplicació de textures que no presentin patrons repetitius a la geometria generada és un punt clau del projecte.

8.3 Abast

El projecte consisteix és la creació de una aplicació que permeti visualitzar i navegar per terrenys generats proceduralment. Aquests terrenys haurien de ser potencialment infinits, és a dir, l'usuari pot explorar l'entorn en una direcció arbitraria de forma indefinida i per tant, degut a que tenim recursos finits, la generació d'aquestos s'ha de realitzar sota demanada. Un altre aspecte que inclourem al projecte és el de la texturització de la orografia generada que hauria de proporcionar més realisme a l'escena.

En les següents subseccions descriurem amb més detall els tres aspectes que engloben la totalitat del projecte indicant les tècniques i tecnologies que es tenen pensades per a la realització del mateix.

8.3.1 Generació de la geometria

Aquesta és la part principal del projecte. Consisteix en generar geometria que representi terrenys de forma que un cop generada es pugui visualitzar i explorar. Aquesta geometria, com hem esmentat anteriorment, ha de poder generar-se a mesura que es vagi necessitant de forma que doni la sensació de que es pot avançar per l'escena indefinidament. Per a realitzar aquesta tasca s'utilitzaran diferents funcions de generació de soroll. Aquestes funcions normalment tenen un comportament aleatori cosa que ens planteja un problema de coherència. S'ha de garantir que la orografia generada per una regió concreta de l'espai sigui sempre la mateixa. A més l'aplicació hauria de poder generar diferents tipus de terrenys ja sigui per voluntat de l'usuari o depenent de l'espai ocupat.

Un altre aspecte que hauríem de tenir en compte és el de la eficiència, cal aconseguir un frame-rate prou alt com per a que la navegació en l'escena sigui fluida. Per aconseguir-ho, en primer lloc, cal notar que aquest tipus de processos són molt paral·lelitzables i per tant hauríem d'intentar explotar-ho utilitzant algun dels paradigmes de programació paral·lela que s'ofereixen. Per una altra banda tenim la part gràfica. És evident que les parts del terreny que es trobin molt lluny de la posició de l'observador no cal que tinguin tant detall com les que estan aprop, per tant també interessa aplicar alguna tècnica que ens permeti utilitzar diferents nivells de detall segons convigui. En aquest cas caldria veure quina s'adapta millor a les nostres necessitats.

8.3.2 Texturat

Per tal de donar cert realisme a l'escena és inprecindibe posar algun tipus de textura. Aquesta textura no només pot simular el color, si no que també pot portar informació sobre on colocar geometria tridimensional, com pot ser la vegetació.

És ben sabut que la repetició sistemàtica de una textura pot provocar patrons perceptibles que poden resultar en una pèrdua de realisme molt notable, per això ens interessa aplicar alguna tècnica que resolgui aquest problema. En la literatura trobem diferents formes de evitar aquest patrons. Una d'aquestes és *wang tiling* la qual, a més de donar bons resultats i ser molt eficient, ens ofereix una de les característiques indispensables per el funcionament del sistema. Aquesta característica és la de que es possible la generació sota demanda en temps real. Això és important ja que, com hem explicat anteriorment, la geometria a la que haurà d'aplicar-se és generada sobre la marxa en temps d'execució de manera que la textura també haurà de ser posada en aquest moment.

Cal dir que aquest tipus de tècniques també utilitzen components aleatòries de forma que tenim el mateix problema de coherència que teníem amb la geometria. També cal remarcar que la tècnica proposada pot ser força complicada de implementar en el context en el que estem i per tant pot ser que s'hagin de tenir en compte alternatives.

8.3.3 CUDA

Com hem vist anteriorment ens és molt interessant utilitzar una plataforma de programació paral·lela a l'hora de realitzar aquest projecte. CUDA ens proporciona un mecanisme per utilitzar els nuclis de les targes gràfiques per a realitzar còmputs en paral·lel. D'aquesta forma no només utilitzem la potència de una gran quantitat de processadors si no que a més, estalviem un gran nombre de ample de banda entre la CPU i la GPU.

Donat que CUDA es propietat de NVIDIA només podrem utilitzar aquesta tecnologia en targes d'aquesta marca i per tant estem limitats en aquest sentit. Tot hi que existeixen altres plataformes que ofereixen més compatiblitat hem escollit CUDA per motius de interès personal en la tecnologia.

8.4 Metodologia i rigor

Per tal de fer el seguiment del projecte, s'establiran reunions periòdiques amb el director del projecte per tal de assegurar que tota la feina va per bon camí. També caldrà veuré que la qualitat del software creat és prou bona. En aquest sentit caldrà veure que l'aplicació permet navegar pel terreny de forma fluida en hardware de gamma mitja i les textures no creen patrons identificables. Aquestes parts poden ser subjectives i per tant seria bo crear algun tipus d'experiment amb usuaris que ens confirmi empíricament que realment els resultats són correctes.

Part I

Planificació del projecte

8.1 Introducció

En aquest document descriurem la planificació del projecte titulat "Generació procedural de terrenys potencialment infinits utilitzant CUDA™". Concretament farem una descomposició de tasques indicant les possibles dependències entre elles i fent una estimació de el temps necessari per a la realització d'aquestes. Hem dividit les tasques en diferents etapes que no tenen per que ser dependents però que segUeixen un ordre lògic. La dependència entre tasques s'hauria de poder extreure del context en les respectives descripcions, tot hi això s'adjunta en la ultima secció un diagrama que representa la dependència de tasques. Les etiquetes d'aquests diagrama fan referència a els nombres entre parèntesi que hi ha al final del títol de cada tasca. Per ultim és mostra el diagrama de GANTT del projecte.

8.2 Aprenentatge i recerca

Aquesta etapa consisteix en adquirir els coneixements necessaris per a la realització del projecte. Està dividida en tres tasques corresponents a els camps principals tractats. En aquesta etapa els recursos que és necessiten són bàsicament les publicacions.

8.2.1 Recerca sobre la generació procedural de terrenys (1)

Aquesta tasca consisteix en fer recerca sobre els treballs realitzats sobre la matèria de generació procedural de terrenys. Al finalitzar aquesta tasca s'hauria de tenir una visó clara de que és pot fer i tenir una idea de quines tècniques s'adaptaran millor a les nostres necessitats.

El temps esperat per a la realització és de 15 dies. No s'esperen contratemps.

8.2.2 Recerca sobre la generació de textures acícliques (2)

Aquesta tasca consisteix en fer recerca sobre els treballs realitzats sobre la matèria de generació de textures acícliques. Al finalitzar aquesta tasca, s'hauria de tenir una visió clara de les possibilitats que tenim en aquest tema i tenir una idea de quines s'adaptarien millor a les nostres necessitats.

El temps esperat de realització és de 15 dies. No s'esperen contratemps.

8.2.3 Adquisició de coneixements de CUDA (3)

Aquesta tasca consisteix en adquirir els coneixements necessaris de CUDA per tal de poder valorar tenint en compte la tecnologia les tècniques trobades en (1) i (2). Aquest coneixements també han de servir per a la realització el projecte.

El temps esperat de realització és de 15 dies. No s'esperen contratemps.

8.3 Adaptació de les tècniques a les necessitat del projecte

En aquesta etapa consisteix en, un cop assimilats els conceptes necessaris per a la realització del projecte, adaptar-los de forma que es puguin utilitzar en el context en el que estem treballant. No s'utilitzaran més recursos que els temporals.

8.3.1 Adaptació de les tècniques de generació de terrenys escollides (4)

Aquesta tasca consisteix en buscar formes d'adaptar segons convingui les tècniques de generació de terrenys escollides en (1). Al finalitzar aquesta tasca, s'haurà de haver vist alguna forma de implementar les tècniques amb CUDA de forma que puguin ser usades sota demanda.

El temps esperat de realització és de 7 dies. No s'esperen contratemps.

8.3.2 Adaptació de les tècniques de generació de textures escollides (5)

Aquesta tasca consisteix en buscar formes d'adaptar segons convingui les tècniques de generació de textures escollides en (2). Al finalitzar aquesta tasca, s'haurà de haver vist alguna forma de implementar les tècniques amb CUDA de forma que puguin ser usades sota demanda.

El temps esperat de realització és de 7 dies. No s'esperen contratemps.

8.4 Primer prototipus

En aquesta etapa s'abordarà la programació de una primera versió del software que permet visualitzar els resultats obtinguts. Els recursos necessaris per a la realització d'aquesta etapa, a part dels temporals, són un un equip que suporti CUDA^{TM}.

8.4.1 Programació de un visualitzador simple (6)

Aquesta tasca consisteix en la programació de un visualitzador senzill per a poder mostrar de forma preliminar els resultats. Aquest visualitzador hauria d'oferir un mínim de interacció i tenir les funcions bàsiques per poder valorar d'alguna forma el rendiment i el resultat visual obtingut.

El temps esperat de realització és de 7 dies. No s'esperen contratemps.

8.4.2 Programació de una primera versió de la generació de terrenys (7)

Aquesta tasca consisteix en fer una primera implementació en CUDA de algun dels algorismes de generació de terreny. Al finalitzar aquesta tasca s'hauria de poder visualitzar amb el visualitzadors de (6) la geometria generada.

El temps esperat de realització és de 15 dies. Aquesta tasca, degut a al seva complexitat, podria ser que s'allargués més del compte. Per això afegirem una tasca extra per tal de tenir en compte aquest tipus de retards.

8.4.3 Programació de una primera versió de la generació de textures (8)

Aquesta tasca consisteix en fer una primera implementació utilitzant CUDA de la tècnica de generació de textures acícliques escollida. Al final d'aquesta tasca s'hauria de poder visualitzar la geometria generada a (7) texturada de forma bàsica. Pot passar que la tècnica escollida sigui

massa complicada de implementar en el context en el que treballem i s'hagin de tenir en compte alternatives cosa que podria retardar el pla.

El temps esperat de realització és de 15 dies. Aquesta tasca, degut a la seva complexitat, podria ser que s'allargués més del compte. Per això afegirem una tasca que modeli aquest temps extra. Cal notar que aquesta tasca serà la mateix que la que utilitzem en (8) per al mateix propòsit.

8.4.4 Finalització de la primera etapa (9)

Aquesta tasca vol modelar el temps extra que puguin suposar les complicacions trobades a (7) i (8).
El temps estimat màxim que haurien de suposar aquest tipus de retards no ha de superar els 7 dies.

8.5 Avaluació dels resultats preliminars

En aquesta etapa es valoraran els resultats obtinguts en l'etapa anterior per tal de veure si els passos seguits donen els resultats desitjats. Els recursos que es necessiten en aquesta etapa són, a part dels temporals, un equip compatible amb CUDA^{TM}.

8.5.1 Avaluació del rendiment del primer prototipus (10)

Aquesta tasca consisteix en avaluar el rendiment del primer prototipus creat en (6), (7) i (8). Al finalitzar aquesta tasca s'haurà de haver valorat si el rendiment obtingut en diferents situacions de l'execució del prototipus és el desitjat. Si el resultat fos inferior a l'esperat, caldria replantejar decisions preses anteriorment cosa que retardaria notablement el projecte.

El temps esperat de realització és de 7 dies. No s'esperen contratemps.

8.5.2 Avaluació del resultat visual del primer prototipus (11)

Aquesta tasca consisteix en avaluar el resultat visual obtingut amb el primer prototipus creat en (6), (7) i (8). Al finalitzar aquesta tasca s'hauria de haver valorat si el resultat és el desitjat. En cas que no ho sigui caldrà optar per la utilització de tècniques diferents cosa que pot comportar un retràs considerable al projecte.

El temps esperat de realització és de 7 dies. No s'esperen contratemps.

8.5.3 Refinament del primer prototipus (12)

Aquesta tasca consisteix en utilitzar els resultats de la avaluació del projecte per tal de millorar els aspectes no satisfactoris que es puguin presentar. Al finalitzar aquesta tasca el resultat visual com de rendiment del primer prototipus han de ser els esperats.

El temps d'aquesta tasca pot ser molt variable depenent dels resultats obtinguts a (9) i (10). El temps màxim no hauria de superar els 15 dies.

8.6 Versió final

Aquesta etapa consisteix en refinar el primer prototipus per tal de que quedi una versió del software presentable. Els recursos que es necessiten en aquesta etapa són, a part dels derivats dels temporals, un equip compatible amb CUDA^{TM}.

8.6.1 Versió final del visualitzador (13)

Aquesta tasca consisteix en la millora del visualitzador per tal de que incorpori totes les funcionalitats requerides i tingui una usabilitat suficientment correcta.

El temps esperat de realització és de 7 dies. No s'esperen contratemps

8.6.2 Versió final de la generació de terreny (14)

Aquesta tasca consisteix en programar la versió final dels algorismes de generació de terrenys introduint tota mena de optimitzacions i millores visuals de forma que els resultats en el camp de generació de terreny siguin satisfactoris i finals.

El temps esperat de realització és de 15 dies. No s'esperen contratemps.

8.6.3 Versió final de la generació de textures (15)

Aquesta tasca consisteix en programar la versió final de les tècniques de generació de textures acícliques. En aquest camp s'espera sobretot millores visuals.

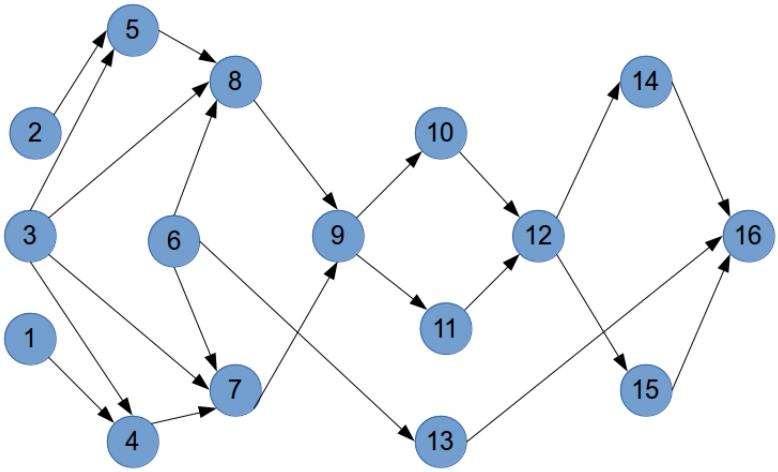
El temps esperat de realització és de 15 dies. No s'esperen contratemps.

8.7 Redactat de la memòria (16)

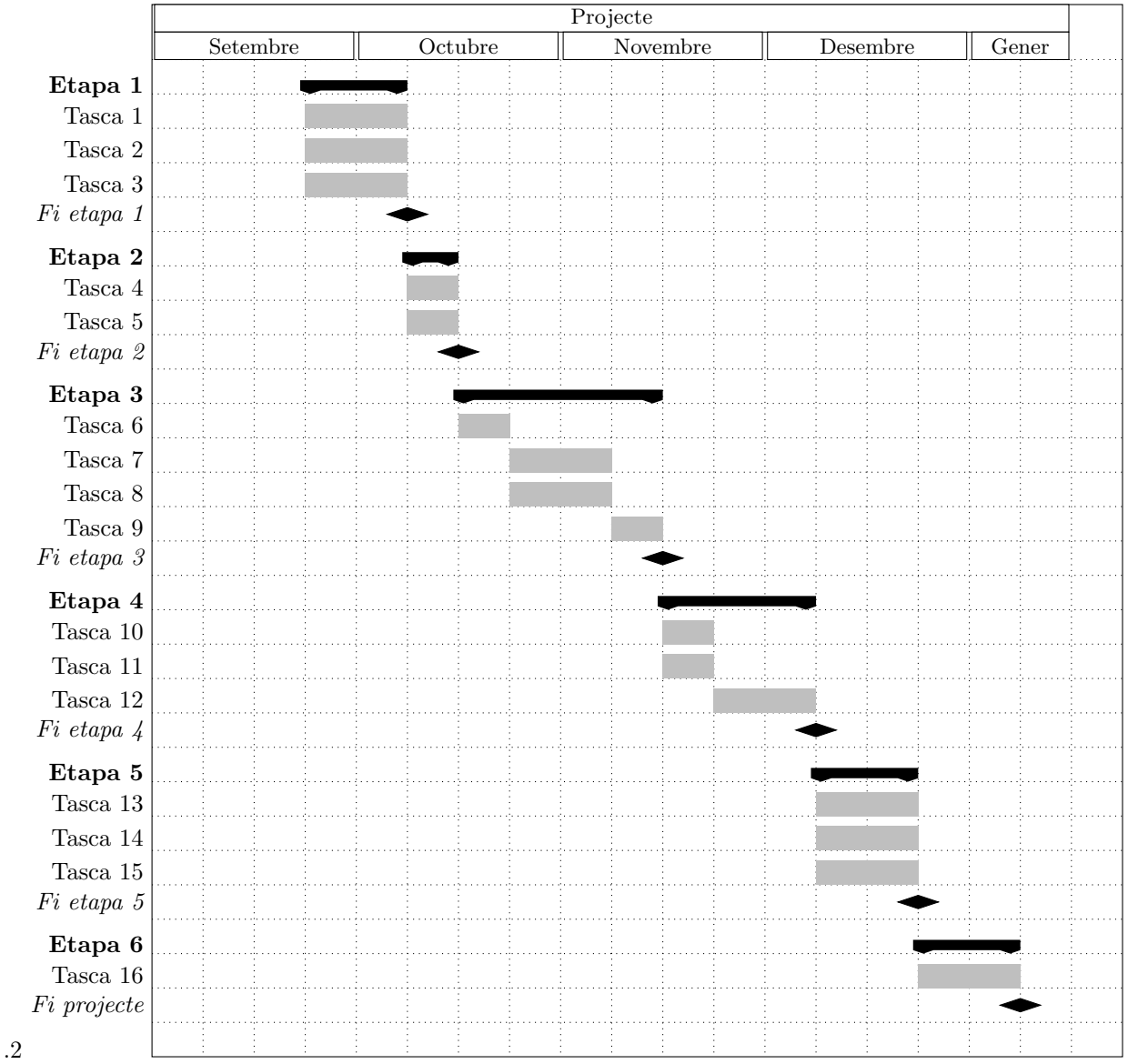
En Aquesta etapa només hi ha una tasca i per tant la tractarem com a una tasca. Aquesta tasca consisteix en la redacció de la memòria del projecte. Al finalitzar s'ha de tenir un document que descrigui detalladament tots els passos seguits per a la realització del projecte.

El temps esperat de realització és de 15 dies. No s'esperen contratemps.

8.8 Diagrama de dependències



8.9 Diagrama de GANTT



.2

Part II

Pressupost i sostenibilitat

8.1 Introducció

En aquest document mostrarem el pressupost del projecte titulat "Generació procedural de terrenys potencialment infinits utilitzant CUDA". Els costos d'aquest projecte no seran gaire diferents dels de un curs acadèmic normal (costos de matricula, costos de lloguer de pis, desplaçaments...). Per aquest motiu hem decidit simular ser una petita empresa. Aquesta empresa només té un treballador i disposa de una oficina.

8.2 Identificació dels costs

Com hem dit anteriorment, per tal de que l'exercici tingui més sentit, simularem ser una empresa petita amb una oficina. És evident que els primers costos que podem veure són els relacionats amb aquesta oficina. En primer lloc el lloguer, seguit de les despeses de llum, aigua i assegurança que pugui generar i el mobiliari que hagem de menester(taules , cadires, etc). Un altre aspecte a tenir en compte és el del maquinari que utilitzarem. Caldrà com a mínim un PC amb una tarja gràfica compatible amb NVIDIA CUDA^{TM}amb tots els seus perifèrics. També podríem tenir en compte el cost de alguns llibres i publicacions referents en la matèria i material de oficina (tòner, paper, etc.). La empresa té un treballador que s'encarrega de tot per tant també hem de tenir en compte el seu sou.

8.3 Estimació dels costos

8.3.1 Costos de la oficina

Com hem vist abans podem desglossar els costos que comporta l'oficina en diferents parts. Aquests costos s'estendran durant la totalitat del projecte.

Lloguer

Tenint en compte que només hi ha un treballador, no cal que l'oficina sigui massa gran, per tant com a molt pagarem 700€ de lloguer cada mes.

Despeses de llum, aigua i internet

Donat que a l'oficina només si estarà en unes hores molt concretes, les despeses en llum i aigua no haurien de ser excessivament altes. Podem estimar que aquests costos seran de 100€ mensuals.

Assegurança

L'assegurança ens ha de cobrir qualsevol robatori, o de desastre que pugues passar. El cost estimat és de uns 300€ anuals. Cal dir que aquesta xifra és totalment inventada ja que desconeixem per complet quin cost real pot tenir una assegurança d'aquest tipus i caldria demanar un pressupost a una agència de assegurances per tenir-ne una idea.

Mobiliari

El mobiliari no cal que sigui massa estens. Una o dos taules, una cadira de oficina, un armari arxivador i una llum de treball. Aquest mobiliari no hauria de costar més de 600€. Cal veure que

aquesta invenció només cal fer-la una vegada però que pot augmentar si es necessita més material o si el mobiliari es deteriora.

8.3.2 Equips

Només necessitem un sol PC de treball ja que només hi ha un treballador. Aquest ha de complir el requisit que ha de tenir una gràfica compatible amb NVIDIA CUDA^{TM}. Aquest tipus de targes gràfiques, depenent del model, poden arribar a valer més de 1000€ però en el nostre cas optarem per un model més discret que valgui uns 500€ ja en tindrem més que suficient. En total el preu de l'equip amb els seus perifèrics ronda els 2000€.

Aquest equip serà imprescindible tenir-lo quan comenci l'etapa 3 del projecte i s'utilitzarà durant tota la resta del projecte.

8.3.3 Material

Material de oficina

Pel que fa a material de oficina (paper, tòner, etc) podem posar-li un cost de 30€ mensuals. Aquest cost s'estendrà durant la totalitat del projecte.

Publicacions

Aquest projecte té una part molt important basada en la recerca i l'aprenentatge de noves tecnologies per tant de ben segur que haurem de consultar publicacions de diferents tipus. Per a aquest aspecte creiem que amb uns 300€ en tindrem suficient.

Aquesta despesa es tindrà només començar el projecte ja que són recursos necessaris en la primera etapa.

8.3.4 Salaris

És evident que el treballador ha de tenir un salari. Hem determinat que un sou de 1500€ mensuals és suficient. Aquest salari s'haurà de pagar durant la totalitat del projecte.

8.3.5 Imprevistos

Com a imprevistos posarem 1000€. Aquesta xifra és aproximadament el 10% del costs de 3 mesos de projecte.

8.3.6 Agregat del projecte

Amb el diagrama de Gantt de la part anterior hem pogut veure que com a molt el projecte s'extendrà durant 3 mesos i mitg. A continuació mostrarem l'agregat del projecte per aquesta durada.

Partida	Cost
Oficina	
Lloguer	700€/mes
Despeses diverses	100€/mes
Assegurança	300€/any
Mobiliari	600€
Subtotal per a 3.5 mesos	3488€
Equips	
PC de treball	1400€
Perifèrics diversos	600€
Subtotal per a 3.5 mesos	2000€
Material	
Material de oficina	30€/mes
Publicacions	300€
Subtotal per a 3.5 mesos	405€
Salaris	
Salari treballador	1500€/mes
Subtotal per a 3.5 mesos	5250€
Imprevistos	
Despeses imprevistes	1000€
Subtotal per a 3.5 mesos	1000€
Total per a 3.5 mesos	12143€

8.4 Viabilitat econòmica

Si tenim en compte que la duració màxima del projecte és de 3 mesos i mitg, veiem que la inversió que s'ha de fer total és de uns 12000€. Pel que fa a l'aplicació que estarem desenvolupant durant el projecte no creiem que sigui un producte que algú estigui interessat a comprar si no que és més com una mostra de les possibilitats de la tecnologia. El que si que podríem vendre és la integració de la tecnologia a altres sistemes, cosa que podria fer rentable la inversió. El tipus de tecnologia que estem desenvolupant pot ser molt interessant en el mon dels videojocs que, com se sap, és una industria que va a la alça i per tant tenim més motius per pensar que hi pot haver interès pel nostre treball.

8.5 Impacte social i ambiental

Pel que fa a l'impacte social i ambiental no creiem que sigui massa notable, al menys directament. No generarem llocs de treball ni facilitarem la vida a ningú. Tampoc empitjorarem ni millorarem el medi ambient de cap forma que pugui ser significativa.

8.6 Rendició de costos i control

Donat que esperem un impacte mediambiental negligible, no té massa sentit proposar diferents mecanismes de rendició de comptes per a quantificar els efectes en sostenibilitat. L'únic aspecte

en el que podem ser sostenibles és en la gestió de l'oficina intentant gastar el mínim de electricitat possible i reciclar el material de oficina. Aquestes mesures es veuen reflectides en les despeses, és a dir, com més les apliquem menys gastem per tant, podríem posar límits en aquest tipus de costos per tal de tenir-ne un control.

Part III

Context i bibliografia

8.1 Introducció

En aquest document mostrarem quin és el context i l'estat de l'art referent al projecte titulat "Generació procedural de terrenys potencialment infinits utilitzant CUDA". Per tal de fer referència a les múltiples entrades bibliogràfiques que presentarem utilitzarem la eina BibTeX.

8.2 Context

En aquesta secció mostrarem a quins sectors podria interessar la feina realitzada durant el projecte. Donat que l'objectiu final no és la creació de un producte que és pugui vendre si no més aviat una demostració del potencial de una tecnologia, els possibles interessats no seran els usuaris finals si no entitats que vulguin integrar aquesta tecnologia en algun dels seus productes. En aquest sentit podem veure que el sector en el que podria tenir més èxit seria el dels videojocs.

8.2.1 Generació procedural de terrenys en els videojocs

En el mon dels videojocs sovint és necessari poder representar grans extensions de terreny navegables per tal de simular un espai obert on el jugador és lliure de fer el que vulgui. Per tal de poder fer això, una de les possibilitats que es presenten és la generació procedural. Aquestes tècniques acostumen a tenenir un cert comportament aleatori cosa que podria conduir a la generació de indrets inaccessibles. Per això la tecnologia introduïda s'hauria de utilitzar per omplir les zones perifèriques o les que no són importants per a la realització dels objectius del joc i que costarien molt temps de construir a mà. Donat que aquestes zones possiblement no seran explorades massa freqüentment, no te massa sentit tenir-les guardades a memòria persistent, per tant, la possibilitat de poder generar aquests terrenys sota demanda seria extremadament útil. A més, com que aquests terrenys generats seran els que segurament conformaran els paisatges llunyans, la possibilitat de poder generar geometria amb diferent nivell de detall segons criteris com la distància o la contribució visual també és una característica interessant de cara al rendiment.

Altres camps on és podria aplicar la tècnica són per exemple la generació de planetes en jocs de exploració galàctica. Aquests tipus de jocs podrien permetre explorar l'univers de una forma indefinida generant sistemes planetaris segons convingui. Aquest planetes haurien de poder ser explorats i per tant cal que tinguin una certa orografia. Donat que el nombre de planetes visitats pot ser extremadament gran, i es busca certa varietat, és imprescindible que el relleu sigui generat de forma procedural i sota demanda. A més, la possibilitat de aplicar diferents funcions de generació podria contribuir a la calitat del joc oferint diversos tipus de terrenys. També cal dir que aquí també és necessari utilitzar tècniques que ofereixin diferents nivells de detall ja que pot passar que sigui necessari la visualització de extencions molt grans del planeta (per exemple en l'aproximació al mateix).

8.3 Estat de l'art

Buscant en la literatura sobre el tema de generació de terrenys infinits en temps real trobem multitud de treballs com [Linda07] on es presenta una forma de generar planetes a partir de la subdivisió de una forma geomètrica. En aquest cas no es genera en temps real encara que podem extreure alguna idea interessant de l'enfocament que se li dona.

Un altre dels treballs que s'han fet en el tema és l'eina que es va fer servir per crear el joc SkyCastle. A [Häggström06] diferents formes de generar el terreny de forma procedural. Alguna de les tècniques que es presenten són la del soroll de perlin [Perlin85] o el soroll fractal. També es parla de com texturar el terreny generat evitant patrons repetibles ja sigui amb geometria com amb un simple color.

A [Olsen04] on podem veure com simular en temps real diferents tipus de erosió sobre un terreny generat amb soroll fractal.

Pel que fa a la persistència del terreny generat podem trobar diferents solucions. Una de elles és presentada a [Kelly] on es crea una ciutat infinita que es mostra a mesura que l'usuari la va explorant. Per a fer-ho es divideix el mon en diferents cel·les a les quals la seva posició determina una valor que es passa a una funció de hash. El resultat del hash s'utilitza com a llavor per a un generador de nombres pseudo-aleatoris que determina les característiques de la cel·la.

Tot hi que la majoria del treball trobat intenta explotar en certa mesura GPU és fa difícil trobar alguna font valuosa on s'utilitzi CUDA com a tecnologia. En alguns casos es menciona com a treball futur.

En les següents subseccions parlarem de les publicacions que creiem que ens servirant com a referència sobre el tema en el que treballarem. Bàsicament ens centrarem en els tres camps principals : generació de la geometria, texturació aciclica i CUDA.

8.3.1 Generació de la geometria

La generació procedural de terrenys és un camp de la visió per computador extensament estudiada i per tant en trobem multiples refències en la literatura. En les següents subseccions mostrarem dos referències que ens han inspirat força. Una d'elles es un llibre que parla sobre la generació procedural de forma general [Ebert02] i l'altre és un article trobat en un recull de tècniques avançades de gràfics de renom [Asirvatham05]. En les següents subseccions parlarem amb més detall de que podem trobar en aquestes fonts.

Texturing and Modeling: A Procedural Approach [Ebert02]

En aquest llibre és un referent en la generació procedural de models i textures.Tot hi que podríem pensar que els seus continguts podrien estar desfasats (estem parlant de un llibre del 2002), la majoria d'ells son perfectament aplicables a les nostres necessitat. És cert que caldria adaptar-los a la tecnologia actual però els fonaments i les idees plantejades són perfectament vàlides. El llibre ens ensenya com a partir de .

GPU Gems 2: Chapter 2. Terrain Rendering Using GPU-Based Geometry Clipmaps[Asirvatham05]

Aquest article ens presenta una tècnica molt utilitzada per a la utilització de diferents nivells de detall depenent de la contribució visual. Aquesta tècnica s'inspira en la tècnica de mipmapping de les textures i consisteix bàsicament en tenir generats diferents nivells de detall de tot el ter-

reny i utilitzar el nivell que més convingui. Cal dir que aquesta tècnica pot ser que no sigui la millor en el nostre cas però en podem extreure bones idees a l'hora de fer la nostra implementació.

GPU Gems 2 és un recull de tècniques avançades en gràfics per computador editat per Randima Fernando Program Manager a NVIDIA i Matt Pharr també de NVIDIA en el moment de la publicació.

8.3.2 Texturació aciclica

Wang Tiles for Image and Texture Generation [Cohen03]

Aquest article ens presenta un algorisme eficient i que es pot utilitzar en temps d'execució per a texturar superfícies utilitzant la tècnica de *tiling*. La tècnica presentada té la particularitat que aprofita el comportament acíclic de les *Wang tiles* per tal de evitar formar patrons perceptibles. A més, també presenta formes de generació d'aquest tipus de *tiles* ja sigui en textures 2D o en disposició de geometria 3D.

Aquest article el podem trobar publicat en diferents revistes científiques cosa que ens assegura que la font es de gran qualitat. A més, tots els autors formen part de entitats de renom com pugui ser Microsoft Research, University of Washington o Dresden University of Technology.

8.3.3 CUDA

CUDA by Example [Sanders10]

CUDA by Example mostra com utilitzar cada un dels aspectes de CUDA a través d'exemples. Després de fer una introduicció amb profunditat a la materia, el llibre detalla les tècniques i coses a tenir en compte de les funcionalitats principals de CUDA.

Els autors del llibre són enginyers de NVIDIA que han col·laborat en el disseny de CUDA i per tant són una font extremadament fiable.

Bibliografia

[Perlin85] Ken Perlin. "An Image Synthesizer". A: *SIGGRAPH Comput. Graph.* 19.3 (jul. de 1985), pàg. 287 - 296. ISSN: 0097-8930. DOI: 10.1145/325165.325247. URL: http://doi.acm.org/10.1145/325165.325247.

[Hoppe99] Hugues Hoppe. "Optimization of Mesh Locality for Transparent Vertex Caching". A: *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '99. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1999, pàg. 269 - 276. ISBN: 0-201-48560-5. DOI: 10.1145/311535.311565. URL: http://dx.doi.org/10.1145/311535.311565.

[Efros01] Alexei A. Efros i William T. Freeman. "Image Quilting for Texture Synthesis and Transfer". A: *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '01. New York, NY, USA: ACM, 2001, pàg. 341 - 346. ISBN: 1-58113-374-X. DOI: 10.1145/383259.383296. URL: http://doi.acm.org/10.1145/383259.383296.

[Gribb01] Gil Gribb i Klaus Hartmann. "Fast Extraction of Viewing Frustum Planes from the WorldView-Projection Matrix". A: (2001).

[Ebert02] David S. Ebert et al. *Texturing and Modeling: A Procedural Approach*. 3rd. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2002. ISBN: 1558608486.

[Cohen03] Michael F. Cohen et al. "Wang Tiles for Image and Texture Generation". A: *ACM Trans. Graph.* 22.3 (jul. de 2003), pàg. 287 - 294. ISSN: 0730-0301. DOI: 10.1145/882262.882265. URL: http://doi.acm.org/10.1145/882262.882265.

[Losasso04] Frank Losasso i Hugues Hoppe. "Geometry clipmaps: terrain rendering using nested regular grids". A: *ACM Transactions on Graphics* 23 (2004), pàg. 769 - 776.

[Olsen04] Jacob Olsen. "Realtime procedural terrain generation". A: (2004).

[Wei04] Li-Yi Wei. "Tile-based Texture Mapping on Graphics Hardware". A: *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*. HWWS '04. Grenoble, France: ACM, 2004, pàg. 55 - 63. ISBN: 3-905673-15-0. DOI: 10.1145/1058129.1058138. URL: http://doi.acm.org/10.1145/1058129.1058138.

[Asirvatham05] Arul Asirvatham i Hugues Hoppe. "Terrain Rendering Using GPU-Based Geometry Clipmaps". A: Matt Pharr i Randima Fernando. *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation (Gpu Gems)*. Addison-Wesley Professional, 2005. Cap. 2. ISBN: 0321335597.

[Quilez05] Iñigo Quilez. *float, small and random*. 2005. URL: http://www.iquilezles.org/www/articles/sfrand/sfrand.htm.

[Forsyth06] Tom Forsyth. *Linear-Speed Vertex Cache Optimisation*. 2006. URL: https://tomforsyth1000.github.io/papers/fast_vert_cache_opt.html.

[Häggström06] Hans Häggström. "Real-time generation and rendering of realistic landscapes". Tesi doct. Citeseer, 2006.

[Linda07] Ondrej Linda. *Generation of planetary models by means of fractal algorithms*. Inf. tèc. Czech Technical University, 2007.

[Geiss08] Ryan Geiss. "Generating Complex Procedural Terrains Using the GPU". A: *GPU Gems 3*. Ed. de Hubert Nguyen. Addison-Wesley, 2008, pàg. 7 - 37.

[Castaño09] Ignacio Castaño. *Optimal Grid Rendering*. 2009. URL: http://www.ludicon.com/castano/blog/2009/02/optimal-grid-rendering/.

[Jenkins09] Bob Jenkins. *A hash function for hash Table lookup*. 2009. URL: http://www.burtleburtle.net/bob/hash/doobs.html.

[Sanders10] Jason Sanders i Edward Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. 1st. Addison-Wesley Professional, 2010. ISBN: 0131387685, 9780131387683.

[Giesen11] Fabian Giesen. *A trip through the Graphics Pipeline*. 2011. URL: https://fgiesen.wordpress.com/2011/07/03/a-trip-through-the-graphics-pipeline-2011-part-3/.

[Kubisch15] Christoph Kubisch. *Life of a triangle - NVIDIA's logical pipeline*. 2015. URL: https://developer.nvidia.com/content/life-triangle-nvidias-logical-pipeline.

[NVIDIA17] NVIDIA Corporation. *NVIDIA CUDA C Programming Guide*. 2017.

[Kelly] George Kelly i Hugh McCabe. "Citygen: An Interactive System for Procedural City Generation". A: ().